CAT4OM Integration Manual

Document version: 2026-07-24 (rev 4)

Wire protocol version: `1.0.0`

Audience: developers integrating third-party software with CAT4OM through JSON WebSocket messages.

Scope: CAT4OM server management, group/radio control, message formats, command sequences, state model, ownership model, error handling, and sample clients in C#, Java, Python, and JavaScript.

This manual describes CAT4OM as a black-box CAT engine. It assumes that the developer does **not** reference `CAT4OM.Contracts`, does **not** load CAT4OM assemblies, and communicates with the running service only through JSON over WebSocket.

The current CAT4OM wire protocol is `1.0.0`. The authoritative protocol string sent by the current server is:

```
"protocolVersion": "1.0.0"
```

1. What CAT4OM Is

CAT4OM is a multi-radio CAT control system.

In amateur radio software, **CAT** means **Computer Aided Transceiver**. It is the generic idea that a computer can read and control a radio: frequency, operating mode, VFO selection, split operation, PTT, RIT/XIT, meters, and similar state. Different radio manufacturers expose different CAT protocols: Yaesu CAT, Icom CI-V, Kenwood CAT dialects, Expert Electronics TCI, and others.

CAT4OM sits between external applications and physical or software radios:

```
External client application
        |
        | JSON over WebSocket
        v
CAT4OM Service
        |
        | radio-specific CAT / CI-V / TCI / UDP
        v
Radio or radio software
```

The important design decision is that external applications do **not** need to know the radio-specific CAT frame format. They send high-level JSON commands such as:

```
{
  "type": "request",
```

```
  "id": "req-42",
  "clientId": "server-assigned-client-id",
  "action": "setFrequency",
  "radioId": "ICOM IC-705",
  "params": {
    "frequency": 14074000,
    "vfo": "A"
  }
}
```

CAT4OM translates that high-level request into the correct radio-specific wire command using XML radio handbooks loaded by the service.

2. Mental Model

CAT4OM has two separate WebSocket surfaces:

Surface	Default port	Purpose
Management Port	`5000`	Server configuration, group lifecycle, handbook discovery, serial port discovery, service status.
Control Port	`5001+`	Real-time control of the radios in one running group.

The **Management Port** is global. There is one Management Port for the whole service.

The **Control Port** is group-specific. Each running group has its own Control Port. External CAT clients normally connect to a Control Port after discovering it through the Management Port or through prior configuration.

Example:

```
CAT4OM Service
  Management Port: 5000

  Group: "Main Station"
    Control Port: 5001
    Radios:
      run
      mult

  Group: "Test Bench"
    Control Port: 5002
    Radios:
      ic7300-test
```

An integration can use either surface or both:

Integration type	Uses Management Port	Uses Control Port
A monitoring dashboard	Yes	Optional
A setup/configuration tool	Yes	Optional
A logging program that only controls a radio	Optional	Yes
A complete station controller	Yes	Yes

2.1 Service Process Lifecycle

The WebSocket protocol manages CAT4OM configuration, group lifecycle, radio lifecycle, and radio control. It does not expose a Management action to terminate the CAT4OM Service process.

Current builds run `CAT4OM.Service.exe` as an interactive console process in both installed and portable modes. Operators stop the service from the console with `CTRL+C`; the first `CTRL+C` asks for confirmation and the second `CTRL+C` within 5 seconds requests graceful shutdown. When a terminal close/logoff/shutdown signal is delivered by Windows, CAT4OM requests the same graceful host shutdown internally.

Integrations should treat service process start/stop as deployment or operator responsibility. To stop radio operation through the protocol without terminating the process, use `stopGroup` or `stopRadio`.

3. Transport Rules

CAT4OM uses plain WebSocket text messages.

Rule	Value
Transport	WebSocket
Message type	Text
Payload encoding	UTF-8 JSON
Endpoint path	`/`
Management URL	`ws://<host>:<managementPort>/`
Control URL	`ws://<host>:<controlPort>/`
JSON naming	lower camel case
Message dispatch	Read `type`, then parse the concrete shape
Polymorphic JSON metadata	Not used

Clients should treat JSON property names as case-sensitive. The CAT4OM UI is tolerant in some places, but external integrations should generate the documented lower camel case fields exactly.

3.1 Push-Based Operation: Do Not Poll Actively

CAT4OM is designed as a **push-based** system. A client should open the WebSocket, keep it open, and continuously listen for server messages. The client should not repeatedly poll the server in a tight loop to discover whether something changed.

This is true on both WebSocket surfaces:

Surface	Push message	How it works	What the client should do
Management Port	`serviceStatus`	After the client sends `subscribe`, the server pushes service/group/radio/client status when something changes and also sends periodic keepalives.	Subscribe once, keep reading messages, update local status from each `serviceStatus`.
Control Port	`stateUpdate` and `event`	After connection, the server sends the initial	Keep a receive loop active, update local radio

		radio snapshot in `welcome.radios[]`, then pushes `stateUpdate` whenever radio state changes. It also pushes events such as `ownershipChanged` and `diagnosticsCompleted`.	state from `stateUpdate`, and react to `event` messages.

The correct client model is:

```
connect
send hello
receive welcome
optionally subscribe on Management
keep reading messages forever while connected
send commands only when the user/application needs to change something
update local state from pushed messages
```

The incorrect model is:

```
connect
send hello
call getState every 250 ms
call getServiceStatus every 250 ms
ignore pushed messages
```

Active polling wastes resources, creates unnecessary WebSocket traffic, can make multiple clients noisier than needed, and may cause the client to miss the real ordering of state changes and events. Use explicit read actions such as `getServiceStatus`, `getGroupStatus`, `getAllState`, or `getState` for initial loading, manual refresh, recovery after reconnect, or diagnostics. Do not use them as the primary way to track live state.

For Control clients, this rule is especially important after write commands. A successful `setFrequency` response means the command was accepted/executed by CAT4OM, but the displayed frequency should still be taken from the next `stateUpdate`. The radio may round, reject, delay, or normalize values according to its own protocol and handbook.

4. Required Handshake Pattern

Both Management and Control ports use the same single-step, request/response handshake:

1. Open the WebSocket.
2. Send a `hello` as the **first** message (`protocolVersion`, `appName`, `appVersion`).
3. Read the server reply:
    - on success, a welcome (`managementWelcome` on Management, `welcome` on Control) — validate `type`, `endpoint`, and `protocolVersion`, and store the server-assigned `clientId`;
    - on rejection, an `error` message — the server then closes the connection.
4. Continue with normal commands.

The `hello` presents the client in a single message and the welcome is the acceptance. The server waits up to about 10 seconds for the `hello`; a connection that sends nothing, or whose first message is not a valid hello, receives an `error` and is closed.

*Two ways to reach CAT4OM: this native WebSocket protocol (rich features — multi-radio, push state, ownership), or, for compatible software, the independent **Hamlib NET rigctl command subset emulation** over plain TCP (see §21). This chapter is about the native protocol.*

4.1 The `hello` message

The client sends this as the very first message on either port:

```
{
  "type": "hello",
  "protocolVersion": "1.0.0",
  "appName": "ExampleLogger",
  "appVersion": "1.2.3"
}
```

Field	Required	Meaning
`type`	Yes	Must be `"hello"`.
`protocolVersion`	Yes	Wire protocol version your client speaks. Current value: `1.0.0`. An incompatible value is rejected with `error` code `PROTOCOL_VERSION_MISMATCH`.
`appName`	Yes	Human-readable application name shown in status dashboards, e.g. `Log4OM`, `MyContestLogger`.
`appVersion`	Yes	Your application version string.
`clientInstanceId`	No	Stable identity of your client across connections. Reserved for future reconnection/session handling; accepted and not used today.
`requestMaster`	No	Boolean preference: `true` means "I would like to control" (master). **Reserved**: present in the protocol but **not yet honored** — role assignment is currently "first connection becomes master".
`noRegister`	No	Service observer flag for read-only dashboards and health probes. On Management, `true` receives `managementWelcome` plus `serviceStatus` pushes without being counted as a Management client. On Control, `true` receives live `welcome.radios[]` and `stateUpdate` snapshots without being counted as an interactive Control client or participating in ownership.
`user`	No	Username. Reserved for future authentication; accepted and **not validated** today (intended rules in §4.4).
`password`	No	Password. Reserved for future authentication; accepted and **not validated** today (intended rules in

		§4.4).

`clientId` is **not** chosen by the client — the server generates it and returns it in the welcome.

4.2 Management Port handshake

Connect to the Management Port (usually `ws://host:5000/`), send `hello`, and the server replies with `managementWelcome`.

Phase	Originator	What is sent	Client-side meaning
1	Client	WebSocket connection to `ws://host:managementPort/`.	Accepted if the Management Port is open. Start reading text messages.
2	Client	`hello` (see §4.1).	The first message; presents the client.
3	Management Port	`managementWelcome` on success, or `error` + close on rejection.	Validate `type = "managementWelcome"`, `endpoint = "management"`, `protocolVersion`. Store `clientId`.
4	Client	Normal Management requests (`getServiceStatus`, `subscribe`, `getConfiguration`, `startGroup`, …).	Correlate responses by `id`; handle `serviceStatus` pushes independently. A `noRegister` observer does not send requests: it receives a `serviceStatus` snapshot immediately after welcome and then future pushes.

`managementWelcome` message:

```
{
  "type": "managementWelcome",
  "endpoint": "management",
  "protocolVersion": "1.0.0",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "serviceVersion": "0.1.0.0",
  "authRequired": false
}
```

4.3 Control Port handshake

Connect to a running group's Control Port (e.g. `ws://host:5001/`), send `hello`, and the server replies with `welcome` carrying the assigned `clientId`, the `groupId`, the initial `role`, and a full `radios` snapshot. Normal interactive clients receive `master` or `slave`; service observers that sent `noRegister: true` receive `observer`.

Phase	Originator	What is sent	Client-side meaning
1	Client	WebSocket connection to `ws://host:controlPort/`.	Accepted if the group is running. Start reading text messages.

2	Client	`hello` (see §4.1).	The first message; presents the client.
3	Control Port	`welcome` on success, or `error` + close on rejection.	Validate `type = "welcome"`, `endpoint = "control"`, `protocolVersion`. Store `clientId`, `groupId`, `role`, `radios[]`.
4	Client	Optional `getOwnership` if the client needs write control and is currently `slave`.	`response` with `result.role = "master"` plus an `ownershipChanged` event broadcast to all Control clients.
5	Client	Normal Control requests (`getAllState`, `getState`, `setFrequency`, `setMode`, …).	Correlate responses by `id`; treat `stateUpdate` as authoritative radio state.

`welcome` message:

```
{
  "type": "welcome",
  "endpoint": "control",
  "protocolVersion": "1.0.0",
  "groupId": "main",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "role": "slave",
  "sessionToken": null,
  "radios": [],
  "authRequired": false
}
```

The `clientId` is assigned by the server in the `welcome`. The client then echoes it in Control requests so the server can associate each request with the correct connection, role, and ownership state. Read actions work immediately after the welcome; write actions require master ownership (see §8).

Service observers (`noRegister`)

A Management or Control connection may send `noRegister: true` in its `hello` when it is a dashboard, monitor, or internal health probe that needs authoritative live state but must not participate as a normal client.

Example:

```
{
  "type": "hello",
  "protocolVersion": "1.0.0",
  "appName": "CAT4OM.StatusPage",
  "appVersion": "0.4.0.5",
  "requestMaster": false,
  "noRegister": true
}
```

On the Management Port this creates a **service observer**:

- the server replies with `managementWelcome`;

- the server immediately pushes a `serviceStatus` snapshot;
- the connection receives subsequent `serviceStatus` pushes and keepalives;
- it is not included in `managementClients` or `managementClientList`;
- post-handshake messages are ignored.

On the Control Port this creates a **service observer**:

- the server replies with a normal `welcome`, with `role: "observer"` and a full `radios[]` snapshot;
- the connection receives subsequent `stateUpdate` pushes;
- it is stored separately from regular Control clients and does not appear in `connectedClients` or `clients[]` on Management status;
- it never becomes master, never receives ownership, and is ignored if it sends post-handshake messages;
- it does not keep a group alive for shutdown policies that depend on real Control clients.

Use this mode only for read-only service surfaces. A configuration tool, logging program, UI radio-control page, contest controller, or any application that may send Management or Control requests should connect without `noRegister`.

4.4 Rejection: the `error` message and credentials

If the hello is missing, malformed, or incompatible, the server replies with a connection-level `error` and closes the connection:

```
{
  "type": "error",
  "code": "PROTOCOL_VERSION_MISMATCH",
  "message": "Unsupported protocol version '0.9', expected '1.0.0'."
}
```

Handshake error codes: `MISSING_HELLO` (no/invalid first message), `PROTOCOL_VERSION_MISMATCH`, `MISSING_PARAM` (missing `appName`/`appVersion`), and — in the future — `UNAUTHORIZED`. Unlike a `response`, an `error` carries no `id`: it precedes any request.

Credentials (reserved, not validated today). `user`/`password` in the hello are part of the protocol now, but the server **accepts any value and does not validate them**. When authentication is enabled in the future, these rules will apply — document and follow them now so integrators are ready:

- **`user`** — letters and digits only (`A–Z`, `a–z`, `0–9`), length **3–32**.
- **`password`** — length **8–18**; alphabet = letters + digits + traditional special characters (`! @ # $ % ^ & * ( ) - _ + = . , ?`); must contain **at least** one lowercase letter, one uppercase letter, and one special character.

The `authRequired` flag in the welcome will indicate, in the future, whether credentials are mandatory; today it is informational.

5. Common Message Types

5.1 Request

Client-to-server requests use this JSON shape on both channels:

```
{
  "type": "request",
  "id": "client-generated-correlation-id",
  "clientId": "server-assigned-client-id",
```

```
  "action": "actionName",
  "radioId": "optional-radio-id",
  "params": {
    "name": "value"
  }
}
```

Fields:

Field	Required	Channel	Meaning
`type`	Recommended on Management, required on Control	Both	Always `request`.
`id`	Yes	Both	Client-generated correlation ID. The server echoes it in the response.
`clientId`	No on Management, yes on Control	Both	Server-assigned client ID from the welcome message.
`action`	Yes	Both	Action name, such as `getServiceStatus` or `setFrequency`.
`radioId`	For radio-scoped Control actions	Control	Radio identifier inside the group.
`params`	Action-dependent	Both	JSON object containing action parameters.

Use unique request IDs. A UUID is ideal. Some examples use short IDs for readability, but production clients should avoid reusing IDs while requests are pending.

5.2 Response

Server-to-client responses use this shape:

```
{
  "type": "response",
  "id": "client-generated-correlation-id",
  "success": true,
  "status": "optional-secondary-status",
  "radioId": "optional-radio-id",
  "result": {
    "name": "value"
  }
}
```

Failure response:

```
{
  "type": "response",
  "id": "client-generated-correlation-id",
  "success": false,
  "radioId": "optional-radio-id",
  "result": null,
  "error": {
    "code": "MISSING_PARAM",
```

```
      "message": "groupId is required"
    }
  }
```

Fields:

Field	Meaning
`type`	Always `response`.
`id`	Echoes the request ID. Use it to complete the correct pending request.
`success`	Authoritative outcome flag. Do not infer success from `status`.
`status`	Optional secondary status. Example: `DIAGNOSTICS_STARTED`.
`radioId`	Optional echo of the target radio for radio-scoped actions.
`result`	Action-specific success payload. May be `null`.
`error`	Single error object on failure.
`errors`	Optional array of multiple validation errors, currently used by `setConfiguration` when more than one validation error exists.

`success` is the field to branch on. `status` is not a replacement for `success`.

5.3 Hello

Client-to-server, the **first** message on either port (see §4.1):

```
{
  "type": "hello",
  "protocolVersion": "1.0.0",
  "appName": "ExampleLogger",
  "appVersion": "1.2.3"
}
```

5.4 Error

Server-to-client, connection-level rejection of a `hello` (see §4.4). It has no `id` and the server closes the connection after sending it:

```
{
  "type": "error",
  "code": "PROTOCOL_VERSION_MISMATCH",
  "message": "Unsupported protocol version '0.9', expected '1.0.0'."
}
```

6. Management Port

The Management Port controls the server, not the radio directly. Use it to discover what exists, start/stop groups, inspect configuration, list handbooks, and subscribe to service status changes.

Default URL:

```
ws://localhost:5000/
```

6.1 Management Welcome

After the client sends `hello`, the server replies with:

```
{
  "type": "managementWelcome",
  "endpoint": "management",
  "protocolVersion": "1.0.0",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "serviceVersion": "0.1.0.0",
  "authRequired": false
}
```

Fields:

Field	Meaning
`type`	Must be `managementWelcome`. If you receive `welcome`, you connected to a Control Port instead.
`endpoint`	Must be `management`.
`protocolVersion`	Must be a protocol version your client supports. Current server value is `1.0.0`.
`clientId`	Server-assigned client ID for this WebSocket connection.
`serviceVersion`	CAT4OM product/service version. This is not the protocol version.
`authRequired`	Indicates that the server configuration contains a management password. Authentication enforcement is reserved for future use.

6.2 Recommended Management Startup Sequence

```
connect ws://host:5000/
send hello
receive managementWelcome
validate endpoint/protocol
send getServiceStatus
optionally send subscribe
use status.groups[] to discover group IDs and control ports
if needed, send startGroup
```

6.3 Management Actions

Action	Purpose	Required params	Success result
`getServiceStatus`	Returns full service status.	none	`ServiceStatus`
`subscribe`	Subscribes to pushed `serviceStatus` messages and returns the current status.	none	`{ "subscribed": true, "status": ServiceStatus }`
`unsubscribe`	Stops pushed `serviceStatus` messages.	none	`{ "subscribed": false }`
`getGroupStatus`	Returns detailed status for one group.	`groupId`	`GroupStatus`
`startGroup`	Starts a configured group and opens its Control	`groupId`	usually no payload

	Port.		
`stopGroup`	Stops a running group and closes its Control Port.	`groupId`	usually no payload
`restartGroup`	Stops and starts one group.	`groupId`	usually no payload
`startRadio`	Starts one configured radio inside an already running group.	`groupId`, `radioId`	`{ "groupId": "...", "radioId": "..." }`
`stopRadio`	Stops one runtime radio inside a running group without deleting configuration.	`groupId`, `radioId`	`{ "groupId": "...", "radioId": "..." }`
`getHandbooks`	Lists radio handbooks available on the service.	none	`{ "handbooks": [ ... ] }`
`getHandbookDetails`	Returns metadata, defaults, VFOs, modes, and validation details for one handbook.	`handbookId`	`HandbookDetails`
`getAvailablePorts`	Lists serial ports on the service host.	none	`{ "ports": [ ... ] }`
`testConnection`	Performs a serial-oriented connection test.	`handbookId`, `serialPort` for serial	`{ "portOpened": true, "radioResponded": bool, "message": "..." }`
`getConfiguration`	Returns the complete server configuration.	none	`Cat4OMConfiguration` JSON
`setConfiguration`	Validates, saves, and reloads the complete server configuration.	`configuration`	`{ "message": "Configuration saved. Restart groups to apply changes." }`
`setLogLevel`	Changes server runtime log level and persists it.	`logLevel`	`{ "logLevel": "normalized-level" }`
`powerOnRadio`	Powers on a radio via a one-shot serial connection (radio must be off; USB serial chip must stay powered from USB bus). Requires `PowerOn` in the handbook.	`groupId`, `radioId`	`{ "message": "..." }`
`powerOffRadio`	Powers off a connected radio in a running group via CAT. The server disconnects the radio immediately after the command is accepted. Requires `PowerOff` in the handbook.	`groupId`, `radioId`	`{ "message": "..." }`

6.4 Get Group Status

`getGroupStatus` returns detailed runtime status for a single group, including its radio list and connected client list.

Request:

```
{
  "type": "request",
  "id": "gstatus-1",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "getGroupStatus",
  "params": {
    "groupId": "main"
  }
}
```

Response shape:

```
{
  "type": "response",
  "id": "gstatus-1",
  "success": true,
  "result": {
    "id": "main",
    "name": "Main Station",
    "controlPort": 5001,
    "autoStart": true,
    "isRunning": true,
    "radioCount": 1,
    "connectedClients": 1,
    "clients": [
      {
        "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
        "remoteEndpoint": "127.0.0.1:53100",
        "connectedAt": "2026-06-02T20:16:00Z",
        "role": "master",
        "appName": "ExampleLogger",
        "appVersion": "1.2.3",
        "protocolVersion": "1.0.0",
        "isSubscribed": false
      }
    ],
    "radios": [
      {
        "radioId": "run",
        "radioName": "Run Radio",
        "connectionState": "connected",
        "lastSeen": "2026-06-02T20:16:30Z",
        "activeVfo": "A",
        "frequency": 14074000,
        "mode": "USB"
      }
    ]
  }
}
```

If the group does not exist, the response has `success: false` with code `NOT_FOUND`.

> ***Follower radios in status.*** *Unlike the Control Port, Management status* ***includes*** *Follower radios (those that mirror another radio). A radio that participates in a Leading/Follower relationship carries*

an extra optional `follow` block. It is present only for radios in a link and never appears on the Control Port:

`jsonc

{

"radioId": "sdr-rx",

"radioName": "SDR receiver",

"connectionState": "connected",

"follow": {

"role": "follower", // "follower" or "leading"

"leadingRadioId": "run", // set on followers

"followerRadioIds": ["sdr-rx"],// set on leadings

"lastForwardAction": "frequency",

"lastForwardSuccess": true,

"lastForwardSuccessAt": "2026-06-02T20:16:31Z",

"forwardErrorCount": 0,

"lastSkipReason": null, // e.g. "MODE_NOT_SUPPORTED" when a mode could not be mirrored

"bidirectional": false, // true when this follower also reverses frequency (Follower dial -> Leading)

"lastReverseAction": null, // present only when bidirectional is true and at least one reverse attempt occurred

"lastReverseSuccess": null,

"lastReverseSuccessAt": null,

"reverseErrorCount": 0,

"lastReverseSkipReason": null // e.g. "TX_IN_PROGRESS" when the Leading was transmitting

}

}

`

The `bidirectional` and `lastReverse*/reverseErrorCount` fields are present only on a follower's block (never on a leading's), and only describe the reverse leg (Follower physical dial change -> Leading). Mode is never reversed, only frequency. Use this block for diagnostics/monitoring only; Follower radios are not commandable and are not exposed on the Control Port.

6.5 Service Status Push

After `subscribe`, the server sends `serviceStatus` messages:

```
{
  "type": "serviceStatus",
  "status": {
    "version": "0.1.0.0",
```

```
    "uptime": "00:12:34.5678901",
    "managementPort": 5000,
    "handbooksPath": "C:\\Users\\operator\\AppData\\Local\\CAT4OM\\handbooks",
    "configurationFile": "C:\\Users\\operator\\AppData\\Local\\CAT4OM\\cat4om-
server.json",
    "logPath": "C:\\Users\\operator\\AppData\\Local\\CAT4OM\\log",
    "managementClients": 1,
    "managementClientList": [
      {
        "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
        "remoteEndpoint": "127.0.0.1:53000",
        "connectedAt": "2026-06-02T20:15:00Z",
        "appName": "ExampleManager",
        "appVersion": "1.0.0",
        "protocolVersion": "1.0.0",
        "isSubscribed": true
      }
    ],
    "groups": [
      {
        "id": "main",
        "name": "Main Station",
        "controlPort": 5001,
        "autoStart": true,
        "isRunning": true,
        "radioCount": 1,
        "connectedClients": 2,
        "clients": [
          {
            "clientId": "control-client-id",
            "remoteEndpoint": "192.168.1.40:53100",
            "connectedAt": "2026-06-02T20:16:00Z",
            "role": "master",
            "appName": "ExampleLogger",
            "appVersion": "1.2.3",
            "protocolVersion": "1.0.0",
            "isSubscribed": false
          }
        ],
        "radios": [
          {
            "radioId": "run",
            "radioName": "Run Radio",
            "connectionState": "connected",
            "lastSeen": "2026-06-02T20:16:30Z",
            "activeVfo": "A",
            "frequency": 14074000,
            "mode": "USB"
          }
        ]
      }
    ]
  },
```

```
  "isKeepAlive": false,
  "timestamp": "2026-06-02T20:16:35Z"
}
```

The server sends status pushes when the service state changes. It also sends a keepalive about every 15 seconds to subscribed clients. When `isKeepAlive` is `true`, the message may contain the same status as before; clients should still accept it as proof that the WebSocket is alive.

6.6 Discover Groups

Request:

```
{
  "type": "request",
  "id": "status-1",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "getServiceStatus",
  "params": {}
}
```

Response:

```
{
  "type": "response",
  "id": "status-1",
  "success": true,
  "result": {
    "version": "0.1.0.0",
    "uptime": "00:01:05.1234567",
    "managementPort": 5000,
    "handbooksPath": "C:\\Users\\operator\\AppData\\Local\\CAT4OM\\handbooks",
    "configurationFile": "C:\\Users\\operator\\AppData\\Local\\CAT4OM\\cat4om-
server.json",
    "managementClients": 1,
    "managementClientList": [],
    "groups": [
      {
        "id": "main",
        "name": "Main Station",
        "controlPort": 5001,
        "autoStart": true,
        "isRunning": true,
        "radioCount": 1,
        "connectedClients": 0,
        "clients": [],
        "radios": [
          {
            "radioId": "run",
            "radioName": "Run Radio",
            "connectionState": "connected",
            "lastSeen": "2026-06-02T20:00:00Z",
            "activeVfo": "A",
            "frequency": 14074000,
            "mode": "USB"
          }
        ]
      }
```

```
      ]
    }
  }
```

Use `groups[].controlPort` to know where a Control client should connect.

6.7 Start A Group

If a group is configured but not running, its Control Port is not open. Start it first:

```
{
  "type": "request",
  "id": "start-main",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "startGroup",
  "params": {
    "groupId": "main"
  }
}
```

Success:

```
{
  "type": "response",
  "id": "start-main",
  "success": true,
  "result": null
}
```

Failure example:

```
{
  "type": "response",
  "id": "start-main",
  "success": false,
  "error": {
    "code": "START_FAILED",
    "message": "Failed to start group: main"
  }
}
```

6.8 Start Or Stop One Radio

`startRadio` and `stopRadio` operate on one radio inside a running group. They do not edit the saved configuration.

Start one radio:

```
{
  "type": "request",
  "id": "start-radio-run",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "startRadio",
  "params": {
    "groupId": "main",
    "radioId": "run"
  }
}
```

Stop one radio:

```
{
```

```
  "type": "request",
  "id": "stop-radio-run",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "stopRadio",
  "params": {
    "groupId": "main",
    "radioId": "run"
  }
}
```

When a radio is stopped, Management status exposes it as:

```
{
  "radioId": "run",
  "radioName": "Run Radio",
  "connectionState": "stopped"
}
```

6.9 Handbook Discovery

CAT4OM uses XML handbooks to describe radio-specific protocols. External clients do not need to parse XML, but they can use handbook discovery to build configuration UIs.

List handbooks:

```
{
  "type": "request",
  "id": "handbooks-1",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "getHandbooks",
  "params": {}
}
```

Response shape:

```
{
  "type": "response",
  "id": "handbooks-1",
  "success": true,
  "result": {
    "handbooks": [
      {
        "id": "icom/ic7300",
        "manufacturer": "Icom",
        "model": "IC-7300",
        "version": "1.0.0",
        "interfaceType": "Serial",
        "protocol": "CI-V",
        "vfoCount": 2,
        "modeCount": 12
      }
    ]
  }
}
```

Get details:

```
{
  "type": "request",
  "id": "handbook-details-1",
```

```
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "getHandbookDetails",
  "params": {
    "handbookId": "icom/ic7300"
  }
}
```

Typical detail fields:

Field	Meaning
`id`	Handbook identifier used in radio configuration.
`manufacturer`	Manufacturer name from the handbook.
`model`	Radio model from the handbook.
`version`	Handbook version, not CAT4OM program version.
`interfaceType`	`Serial`, `TCI`, or `UDP`.
`protocol`	CAT dialect, for example `CAT`, `CI-V`, or `TCI`.
`availableVfos`	VFO names defined by the handbook.
`supportedModes`	Operating modes exposed by the handbook.
`capabilities`	Informational summary derived from commands. For runtime gating, prefer `RadioState.availableCommands`.
`connectionDefaults`	Default serial/CI-V settings.
`hasConnectionCheck`	Whether the handbook contains a connection check command.
`commandCount`	Number of commands declared by the handbook.
`validationErrors`	Optional handbook validation errors.
`validationWarnings`	Optional handbook validation warnings.

6.10 Configuration JSON

`getConfiguration` returns the complete server configuration. `setConfiguration` replaces the complete configuration; it is not a patch operation.

Minimal shape:

```
{
  "version": "1.0",
  "groups": [
    {
      "id": "main",
      "name": "Main Station",
      "controlPort": 5001,
      "autoStart": true,
      "autoReconnect": true,
      "reconnectIntervalSeconds": 5,
      "shutdownPolicy": "KeepAlive",
      "authentication": {
        "enabled": false,
        "password": null
      },
      "radios": [
```

```
        {
          "id": "run",
          "name": "Run Radio",
          "handbook": "icom/ic7300",
          "enabled": true,
          "connection": {
            "type": "serial",
            "serialPort": "COM3",
            "baudRate": 115200,
            "dataBits": 8,
            "stopBits": 1,
            "parity": "None",
            "civAddress": "94",
            "rtsEnable": true,
            "dtrEnable": true
          },
          "reconnect": {
            "enabled": true,
            "interval": 5000,
            "maxAttempts": 0
          }
        }
      ]
    }
  ],
  "hub": {
    "managementPort": 5000,
    "managementPassword": null,
    "logLevel": "info",
    "stateUpdateInterval": 250
  }
}
```

Connection settings:

Connection type	Required/common fields
`serial`	`serialPort`, `baudRate`, `dataBits`, `stopBits`, `parity`, optional `rtsEnable`, `dtrEnable`, optional `civAddress` for Icom CI-V.
`tci`	`host`, `port`, optional `trxIndex` (default `0`), optional `enableTelemetry` (default `false`).
`udp`	`host`, `port`.
`flex`	`host`, optional `port` (default `4992`), optional `slicePair` (default `"AB"`).

`trxIndex` selects which physical transceiver to control on multi-TRX TCI radios (0 = first). Omit or set to `0` for single-TRX radios. The TRX identity is taken from the index carried by TCI messages, not from the WebSocket connection that happened to receive them. CAT4OM may internally relay TCI sensor telemetry between connections to the same endpoint before applying this TRX filter; integrations still see one independent `RadioState` per configured radio/TRX.

`enableTelemetry` controls whether CAT4OM executes handbook initialization commands marked `RequiresTelemetry`. It defaults to `false`; set it to `true` to request continuous sensor streams such as TCI S-meter, TX power, and SWR updates. Metering read through ordinary polling commands is independent of this option.

`slicePair` selects which pair of SmartSDR slices maps to VFO A and VFO B. Valid values: `"AB"` (default), `"CD"`, `"EF"`, `"GH"`. Use multiple radio configurations pointing to the same host to control different slice pairs independently.

`shutdownPolicy` values:

Value	Meaning
`KeepAlive`	The group stays running and connected after the last client disconnects. Default.
`StopOnLastClient`	The group stops automatically when the last Control client disconnects.

Group-level automatic reconnection fields:

Field	Type	Meaning
`autoReconnect`	bool	Group-level master switch for automatic radio reconnection. Default `true`. When on, every radio in the group whose per-radio `reconnect.enabled` is also `true` is retried automatically if it fails its initial connection (for example the serial port was momentarily busy at start-up) or drops later. When off, no radio in the group is auto-reconnected and recovery requires the `reconnectDisconnectedRadios` action or a group restart.
`reconnectIntervalSeconds`	int (s)	Delay between automatic reconnection attempts for this group's radios. Validated range `1..3600`; a value outside it is rejected with error code `RECONNECT_INTERVAL_RANGE`.

The effective per-radio behaviour is the combination of both levels: a radio is auto-reconnected only when `autoReconnect` (group) **and** `reconnect.enabled` (radio) are both true. The group `reconnectIntervalSeconds` provides the interval; the per-radio `maxAttempts` still applies as a fine-grained cap.

`reconnect` object fields (per radio):

Field	Type	Meaning
`enabled`	bool	Whether this radio participates in automatic reconnection. Effective only when the group `autoReconnect` master switch is also on. Covers both an initial

		connection that never succeeds and a connection that drops later.
`interval`	int (ms)	Per-radio interval between reconnection attempts. Present for backward compatibility; the group `reconnectIntervalSeconds` is the value applied at runtime.
`maxAttempts`	int	Maximum number of reconnection attempts. `0` means retry indefinitely. Applied as a per-radio override on top of the group policy.

Optional `frequencyOffset` object:

```
{
  "frequencyOffset": {
    "enabled": true,
    "vfos": {
      "A": {
        "enabled": true,
        "label": "10 GHz transverter",
        "offsetHz": 10000000000,
        "direction": "add",
        "minFrequencyHz": 10144000000,
        "maxFrequencyHz": 10144200000
      },
      "B": {
        "enabled": false,
        "label": "2.4 GHz transverter",
        "offsetHz": 2400000000,
        "direction": "add",
        "minFrequencyHz": null,
        "maxFrequencyHz": null
      }
    }
  }
}
```

`frequencyOffset` is used for transverter-style operation. The physical radio may tune an IF frequency such as `144100000` Hz, while clients should see and send a higher public frequency such as `10144100000` Hz. CAT4OM keeps the CAT engine on real radio frequencies and applies the offset only at the JSON boundary. The radio-level object is a master switch; the actual offset payload is stored per VFO.

Field	Type	Meaning
`enabled`	bool	Radio-level master switch. When false, all saved VFO offsets remain configured but inactive.
`vfos`	object map	Map of VFO name to VFO-specific offset settings. VFO names are stable identifiers from the handbook/runtime state, not fixed A/B properties.
`vfos[*].enabled`	bool	Enables projection for that VFO.

`vfos[*].label`	string?	Optional display label, for example `10 GHz transverter`.
`vfos[*].offsetHz`	long	Offset in Hz. Must be greater than zero when that VFO is enabled.
`vfos[*].direction`	string	`add` means public frequency = radio frequency + offset. `subtract` means public frequency = radio frequency - offset.
`vfos[*].minFrequencyHz`	long?	Optional minimum public/client frequency accepted by `setFrequency` for that VFO.
`vfos[*].maxFrequencyHz`	long?	Optional maximum public/client frequency accepted by `setFrequency` for that VFO.

Client rule: keep using the normal `frequency` field. Do not invent a separate client-side offset unless your application deliberately wants to display diagnostics. CAT4OM already transforms `frequency` in `welcome`, `stateUpdate`, `getState`, `getAllState`, Management status, and inbound `setFrequency`.

When a frequency offset is active or configured for a VFO, the Control Port includes the VFO-specific `frequencyOffset` object inside that `VfoState`. Treat it as metadata:

- use it to show an operator label such as `10 GHz transverter`;
- use `offsetHz`, `minFrequencyHz`, and `maxFrequencyHz` to choose safe UI input limits;
- do **not** add or subtract it from `vfos[*].frequency`, because those frequencies are already public/client-visible values;
- do **not** subtract it before sending `setFrequency`, because the server performs the inbound conversion.

`hub` fields:

Field	Meaning
`managementPort`	TCP port for the Management Port. Changing this requires a service restart.
`managementPassword`	Reserved for future authentication. Currently unused.
`logLevel`	Persistent server log level. Same values as `setLogLevel`.
`stateUpdateInterval`	Milliseconds between radio state polling cycles. Controls how frequently CAT4OM polls serial/TCI radios for state changes. Default is 250 ms. Lower values produce more responsive updates but increase serial traffic.

Important configuration behavior:

5. `setConfiguration` validates the complete configuration before saving.
6. If validation fails, the response has `success: false`.
7. If multiple validation errors exist, the response may include both `error` and `errors[]`.

8. Runtime safety checks run before saving. Group-level changes and radio membership changes require the group to be stopped. Radio master configuration changes require that specific radio to be stopped/disconnected.
9. The server stops all groups before applying an accepted new configuration.
10. After saving, groups may need to be restarted for changes to take effect.
11. The Management Port itself is read at service startup. Changing `hub.managementPort` generally requires service restart.

Runtime configuration safety:

Change	Required state
Group settings (`name`, `controlPort`, `autoStart`, `shutdownPolicy`, `authentication`)	Group must be stopped.
Add or remove a radio from a group	Group must be stopped.
Radio master configuration (`name`, `handbook`, `enabled`, `connection`, `reconnect`, `frequencyOffset`)	That radio must be stopped/disconnected/unknown. The owning group may still be running if the radio itself is not active.

This is intentionally per-radio: a running group is a container of independent radios. A stopped radio can be edited and restarted without forcing unrelated running radios to be changed.

`setConfiguration` validation errors:

Code	Meaning
`MISSING_PARAM`	The request does not contain a `configuration` field.
`INVALID_CONFIG`	The `configuration` payload cannot be parsed as a CAT4OM server configuration.
`EMPTY_GROUP_ID`	A group has an empty or whitespace-only ID.
`DUPLICATE_GROUP_ID`	Two groups have the same ID, case-insensitively.
`EMPTY_RADIO_ID`	A radio has an empty or whitespace-only ID.
`DUPLICATE_RADIO_ID`	Two radios inside the same group have the same ID, case-insensitively.
`MISSING_RADIO_HANDBOOK`	A radio does not specify a handbook ID.
`MISSING_SERIAL_PORT`	A radio with `connection.type = "serial"` has no serial port configured.
`MGMT_PORT_CONFLICT`	A group control port is equal to the Management Port.
`RECONNECT_INTERVAL_RANGE`	A group `reconnectIntervalSeconds` is outside the allowed range `1..3600`.
`INVALID_FREQUENCY_OFFSET`	A radio has `frequencyOffset.enabled = true` and an enabled VFO offset entry whose `offsetHz` is not greater than zero.
`INVALID_FREQUENCY_OFFSET_DIRECTION`	A radio frequency offset direction is not supported.
`INVALID_FREQUENCY_OFFSET_RANGE`	A radio frequency offset has `minFrequencyHz` greater than `maxFrequencyHz`.
`GROUP_MUST_BE_STOPPED_FOR_CONFIGURATION_CHANGE`	A running group's own settings or radio

	membership would change.
`RADIO_MUST_BE_STOPPED_FOR_CONFIGURATION_CHANGE`	An active radio's master configuration would change.

`setConfiguration` validation warnings:

Code	Meaning
`EMPTY_GROUP_NAME`	A group has no display name. The configuration is still saved.
`EMPTY_RADIO_NAME`	A radio has no display name. The configuration is still saved.

Runtime start validation is separate from save validation. CAT4OM allows two stopped groups to be configured with the same Control Port, but only one running group can bind a given TCP port. When `startGroup` or `restartGroup` detects that another running group already uses the requested Control Port, startup fails with `CONTROL_PORT_IN_USE` in the service logs and the Management action returns `START_FAILED` or `RESTART_FAILED`.

Example `setConfiguration` validation failure:

```
{
  "type": "response",
  "id": "set-config-1",
  "success": false,
  "error": {
    "code": "DUPLICATE_RADIO_ID",
    "message": "Radio ID 'run' is duplicated in group 'main'."
  },
  "errors": [
    {
      "code": "DUPLICATE_RADIO_ID",
      "message": "Radio ID 'run' is duplicated in group 'main'."
    },
    {
      "code": "MISSING_SERIAL_PORT",
      "message": "Radio 'backup' in group 'main' is serial but has no serial port
configured."
    }
  ]
}
```

6.11 Test Connection

`testConnection` is currently serial-oriented. It opens the requested serial port using the selected handbook defaults, then sends the handbook connection check command if one exists.

Request:

```
{
  "type": "request",
  "id": "test-conn-1",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "testConnection",
```

```
  "params": {
    "handbookId": "icom/ic7300",
    "connectionType": "serial",
    "serialPort": "COM3",
    "baudRate": 115200
  }
}
```

Success:

```
{
  "type": "response",
  "id": "test-conn-1",
  "success": true,
  "result": {
    "portOpened": true,
    "radioResponded": true,
    "message": "Radio responded: ..."
  }
}
```

6.12 Log Level

`setLogLevel` changes the server log level immediately and persists it in configuration.

Supported values:

```
trace, debug, info, warn, error, fatal
```

Aliases:

Alias	Normalized value
`information`	`info`
`warning`	`warn`

Request:

```
{
  "type": "request",
  "id": "log-1",
  "clientId": "2d724de1-0aa7-4f97-b785-5ea1d099c221",
  "action": "setLogLevel",
  "params": {
    "logLevel": "debug"
  }
}
```

Response:

```
{
  "type": "response",
  "id": "log-1",
  "success": true,
  "result": {
    "logLevel": "debug"
  }
}
```

6.13 powerOnRadio

`powerOnRadio` powers on a radio that is currently off. The server opens a temporary serial connection using the radio's existing configuration, sends the handbook `PowerOn` sequence, and disposes the connection. No `RadioHandler` is created and no state is published; the radio simply receives the wake-up command.

This action is only meaningful for radios connected via serial (USB or RS-232). The USB serial chip in many radios (e.g., Yaesu FTDX-101D) remains powered from the USB bus even when the radio is in standby, so CAT commands can reach the radio before it is fully on.

Prerequisites:

- `groupId` and `radioId` must identify a configured radio.
- The radio's connection type must be `serial`.
- The handbook must define a `PowerOn` command.

```
{
  "type": "request",
  "id": "pwron-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "powerOnRadio",
  "params": {
    "groupId": "main",
    "radioId": "run"
  }
}
```

Success:

```
{
  "type": "response",
  "id": "pwron-1",
  "success": true,
  "result": {
    "message": "PowerOn command sent to radio 'run'"
  }
}
```

After the radio finishes booting, start it with `startRadio` (or start the group) to establish the full CAT connection.

6.14 powerOffRadio

`powerOffRadio` powers off a connected radio in a running group. The server sends the handbook `PowerOff` command and then immediately disconnects the adapter. The radio transitions to `disconnected` state without waiting for the serial link to go silent.

This is the Management Port equivalent of the Control Port `powerOff` action. Use it from lifecycle automation or from a management client that is not connected to the group's Control Port.

Prerequisites:

- The group must be running.
- The radio must be in `connected` state.
- The handbook must define a `PowerOff` command.

```
{
  "type": "request",
```

```
  "id": "pwroff-mgmt-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "powerOffRadio",
  "params": {
    "groupId": "main",
    "radioId": "run"
  }
}
```

Success:

```
{
  "type": "response",
  "id": "pwroff-mgmt-1",
  "success": true,
  "result": {
    "message": "PowerOff command sent to radio 'run'"
  }
}
```

7. Control Port

The Control Port is where real-time radio control happens. Each running group exposes one Control Port.

Example URL:

```
ws://localhost:5001/
```

You do not connect directly to "radio COM3" or to "radio run". You connect to the group Control Port, then address a specific radio by `radioId` inside JSON requests.

7.1 Control Welcome

After the client sends `hello`, the server replies with:

```
{
  "type": "welcome",
  "endpoint": "control",
  "protocolVersion": "1.0.0",
  "groupId": "main",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "role": "master",
  "sessionToken": null,
  "radios": [
    {
      "radioId": "run",
      "groupId": "main",
      "radioName": "Run Radio",
      "handbookId": "icom/ic7300",
      "manufacturer": "Icom",
      "model": "IC-7300",
      "timestamp": "2026-06-02T20:20:00Z",
      "connectionStatus": "connected",
      "lastSeen": "2026-06-02T20:20:00Z",
      "availableVfos": ["A", "B"],
      "vfos": {
        "A": {
          "vfo": "A",
          "frequency": 14074000,
```

```
            "mode": "USB",
            "filterWidth": 2400
          },
          "B": {
            "vfo": "B",
            "frequency": 7074000,
            "mode": "LSB",
            "filterWidth": 2400
          }
        },
        "activeVfo": "A",
        "txVfo": "A",
        "rxVfos": ["A"],
        "split": false,
        "rit": {
          "enabled": false,
          "offset": 0
        },
        "xit": {
          "enabled": false,
          "offset": 0
        },
        "ptt": false,
        "tuner": "bypass",
        "metering": {
          "sMeter": 0,
          "signalDbm": null,
          "swr": null,
          "power": 0,
          "alc": 0
        },
        "supportedModes": ["LSB", "USB", "CW", "AM", "FM"],
        "availableCommands": ["GetFrequencyA", "GetModeA", "SetFrequency",
    "SetMode", "SetVfo", "SetPtt"],
        "commandVfoScopes": {
          "SetFrequency": "activeOnly"
        },
        "frequencyResolution": 1,
        "frequencyGroups": ["hf"]
      }
    ],
    "authRequired": false
  }
```

For a transverter radio, the same `RadioState` may include active VFO offset metadata and already-projected VFO frequencies:

```
  {
    "radioId": "yaesu/ftdx-101d",
    "radioName": "10 GHz station",
    "vfos": {
      "A": {
        "vfo": "A",
        "frequency": 10007200100,
```

```
        "mode": "USB",
        "filterWidth": 2400,
        "frequencyOffset": {
          "enabled": true,
          "label": "10 GHz transverter",
          "offsetHz": 10000000000,
          "direction": "add",
          "minFrequencyHz": null,
          "maxFrequencyHz": null
        }
      }
    }
  }
```

In this example the physical radio frequency is `7,200,100` Hz, but the Control client displays and sends `10,007,200,100` Hz. The client does not calculate the physical IF frequency; the server does that before dispatching CAT commands.

Fields:

Field	Meaning
`type`	Must be `welcome`.
`endpoint`	Must be `control`.
`protocolVersion`	Current value: `1.0.0`.
`groupId`	Group served by this Control Port.
`clientId`	Server-assigned client ID. Include this in Control requests.
`role`	Initial role: `master`, `slave`, or `observer` for a `noRegister` service observer.
`sessionToken`	Reserved for future use.
`radios`	Full initial radio state snapshot.
`authRequired`	Reserved authentication signal.

> ***Hidden radios (Followers).*** *A group may configure some radios as* ***Followers*** *— radios that automatically mirror another radio's frequency/mode (typically a listening SDR). Follower radios are intentionally* ***absent*** *from every Control-Port surface: they do not appear in this `radios` list, in `stateUpdate`, or in `getAllState`/`getState`, and a command addressed to a follower id returns `RADIO_NOT_FOUND` (identical to an unknown radio). A Control client therefore never needs to know followers exist — the Control protocol is otherwise unchanged. The number of radios in `welcome.radios` may be smaller than the number of radios shown in the Management group status. Followers are visible only on the Management Port (see §6.4).*
>
> ***Service observer role.*** *A `welcome` with `role: "observer"` is returned only to a Control connection that sent `hello.noRegister = true`. Treat it as read-only: use `welcome.radios[]` and future `stateUpdate.radios[]`, but do not send requests or ownership commands.*

7.2 Recommended Control Startup Sequence

```
connect ws://host:controlPort/
send hello
receive welcome
```

```
validate endpoint/protocol
store clientId, groupId, role, radios[]
if you need to control the radio and role is slave:
    send getOwnership
maintain local state from welcome.radios[] and stateUpdate.radios[]
send radio commands only when master
```

The initial `welcome.radios[]` snapshot is already useful state. You do not need to call `getAllState` immediately unless you want a second explicit snapshot after the welcome.

8. Ownership Model

Ownership is one of the most important parts of the Control Port.

CAT4OM allows multiple clients to connect to the same group. For example:

```
CAT4OM UI
Logging program
Remote control tool
Monitoring dashboard (as a service observer)
```

All of them may need to read the same radio state, but it would be dangerous if all of them could transmit write commands at the same time. Two clients changing frequency simultaneously would produce confusing and potentially unsafe station behavior.

Read-only dashboards that are part of CAT4OM itself, or equivalent external health probes, should use `hello.noRegister = true` so they can observe Management and Control streams without affecting client counts or ownership.

CAT4OM solves this with a simple **master/slave** model:

Role	Can read state	Receives pushes	Can send write commands
`master`	Yes	Yes	Yes
`slave`	Yes	Yes	No
`observer`	Yes	Yes	No, and it is not registered as a client

The first client connected to a Control Port becomes master. Later clients become slaves.

If the master disconnects, the server promotes another connected client.

Any connected and identified client can request ownership by sending `getOwnership`. The current server transfers ownership immediately to the requester. The old master becomes slave. All clients receive an `ownershipChanged` event.

8.1 Why The Action Is Called getOwnership

The action name is currently:

```
getOwnership
```

Despite the name, it is not just a query. It actively requests ownership and, when accepted, makes the caller master. Treat it as "become primary controller".

8.2 Request Ownership

Request:

```
{
```

```
  "type": "request",
  "id": "own-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "getOwnership",
  "params": {}
}
```

Response:

```
{
  "type": "response",
  "id": "own-1",
  "success": true,
  "result": {
    "role": "master"
  }
}
```

Event broadcast to all Control clients:

```
{
  "type": "event",
  "event": "ownershipChanged",
  "timestamp": "2026-06-02T20:25:00Z",
  "radioId": null,
  "details": {
    "masterId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38"
  }
}
```

Clients should update their local role by comparing `details.masterId` with their own `clientId`.

8.3 Master-Only Actions

The following actions require master ownership:

```
setFrequency
setMode
setPtt
setVfo
setSplit
setRit
setXit
vfoAtoB
vfoBtoA
vfoEqualize
vfoSwap
sendRaw
sendRawCmdFireAndForget
sendCw
stopCw
setCwSpeed
cwKeyDown
runDiagnostics
cancelDiagnostics
powerOff
reconnectDisconnectedRadios
```

If a slave sends one of these actions, the server responds:

```
{
  "type": "response",
  "id": "freq-1",
  "success": false,
  "error": {
    "code": "NOT_MASTER",
    "message": "Only master client can send commands"
  }
}
```

Read actions do not require master ownership.

8.4 TX Safety Lock

While a radio is transmitting (its `ptt` flag is `true`), the server protects the transmission from any interference that could prevent the radio from receiving the TX-off command. This safety lock is enforced server-side and is independent of the master/slave model.

Two rules apply for as long as `ptt` is `true`:

12. **Ownership is frozen.** A `getOwnership` request is rejected while *any* radio in the group is transmitting. This guarantees that the operator who keyed the transmitter stays master and remains the one able to unkey it. Ownership transfer becomes possible again as soon as the radio returns to RX.
13. **Only the unkey is accepted on the transmitting radio.** For a radio that is transmitting, the master may send only `setPtt` with `enabled = false` (and read-only queries such as `getState`). Every other mutating command (`setFrequency`, `setMode`, `setVfo`, `setSplit`, raw commands, diagnostics, …) is rejected. This keeps the serial bus clear so the TX-off command is delivered reliably.

A rejected request returns:

```
{
  "type": "response",
  "id": "freq-9",
  "success": false,
  "error": {
    "code": "TX_IN_PROGRESS",
    "message": "Radio 'ftdx101d' is transmitting; only PTT-off is accepted until
TX ends."
  }
}
```

Server-side failsafes. The lock cannot strand a radio in transmit:

- If the master client disconnects while a radio is keyed, the server automatically forces `setPtt false` on that radio before promoting a new master.
- A per-radio **TX watchdog** forces `setPtt false` after a maximum continuous transmit duration (default 120 s, hard-capped to 180 s, configurable per radio via `txTimeoutSeconds`). This also recovers the lock if a radio never reports its return to RX.

Client guidance. A control client should treat `TX_IN_PROGRESS` as "the transmitter is busy": disable controls other than unkey while `ptt` is `true`, and rely on `setPtt false` (or releasing the connection) to end transmission. The authoritative `ptt` value always comes from the radio state in `stateUpdate` pushes.

8.5 Limitations

The TX safety lock tracks transmissions that CAT4OM is aware of: PTT initiated over CAT, or TX reported by the radio through polling/push. A transmission keyed purely by external hardware (e.g. a footswitch)

on a radio that does not report its TX status over CAT is not visible to the service and is therefore not covered by the lock.

9. Radio State

`RadioState` is the canonical state object for a radio. It appears in:

Message/action	Location
Control welcome	`radios[]`
`stateUpdate` push	`radios[]`
`getAllState` response	`result.radios[]`
`getState` response	`result`

CAT4OM is push-based. Clients should treat `stateUpdate.radios[]` as authoritative. After sending `setFrequency`, do not assume the frequency changed just because the command response was successful. Use the next pushed state to update your visible model. This matters because CAT4OM talks to real radios, and real radios can reject, delay, normalize, or independently change state.

Frequency semantics for clients: `vfos[*].frequency` is always the public/user-visible frequency in Hz for that VFO. For ordinary VFOs this equals the physical radio frequency. For VFOs with an active `frequencyOffset`, CAT4OM transforms the physical radio frequency before publishing it and transforms inbound `setFrequency.params.frequency` back to the physical radio frequency before commanding the radio.

`RadioState` does not expose a single radio-wide `frequencyOffset`. Offset metadata is VFO-scoped and lives in `vfos[*].frequencyOffset`. Its presence does not change how clients read or write `frequency`; it only explains the projection and gives UI clients enough metadata to label the VFO and enforce public-frequency input limits.

9.1 RadioState Fields

Field	Type	Meaning
`radioId`	string	Stable radio ID inside the group. Use this in radio-scoped requests.
`groupId`	string	Group that owns this radio state.
`radioName`	string	Human-readable configured name.
`handbookId`	string	Handbook identifier used by the service.
`manufacturer`	string	Manufacturer from the active handbook.
`model`	string	Model from the active handbook.
`timestamp`	DateTime string	UTC timestamp for this state snapshot.
`connectionStatus`	string	Connection state. See below.
`lastSeen`	DateTime string	Last time the radio produced valid activity/state.
`availableVfos`	string array	VFO tokens exposed by the handbook, such as `A`, `B`, `C`, `MAIN`, `SUB`.

`vfos`	object map	Per-VFO state keyed by VFO name.
`activeVfo`	string	VFO currently selected for operator commands.
`txVfo`	string	VFO used for transmit. Usually same as `activeVfo`, different in split.
`rxVfos`	string array	VFOs currently receiving. Dual-watch radios may report more than one.
`split`	bool	Whether CAT4OM considers the radio in split operation.
`rit`	object	Receiver incremental tuning state.
`xit`	object	Transmitter incremental tuning state.
`ptt`	bool	Push-to-talk state. `true` means transmitting. Also set to `true` during CW transmissions initiated via `sendCw` or `cwKeyDown(enabled: true)`, so the TX safety lock covers CW keying.
`cwWpm`	int?	Current CW keyer speed in words per minute, as last reported by the radio. `null` when the radio has not yet reported a speed. Updated by TCI `CW_MACROS_SPEED` push messages. Absent from the wire when `null`.
`legacyControlled`	bool	`true` when this radio is currently under exclusive control through the Hamlib NET rigctl command subset (see §21). While `true`, native clients still receive its state but mutating commands against it are rejected with `RADIO_CLAIMED`. Clients may show an external-control indicator and disable controls for this radio.
`tuner`	string	Tuner state, when known. Common values: `bypass`, `active`, `tuning`. May be `null` if the radio does not report tuner state.
`metering`	object	Meter readings such as S-meter, SWR, power, ALC.
`supportedModes`	string array	Modes supported by the active handbook.
`availableCommands`	string array	Actual command names declared by the handbook. Use for feature gating.
`commandVfoScopes`	object map	Sparse map of command-specific VFO targeting restrictions.
`frequencyResolution`	int	Frequency resolution in Hz.
`frequencyGroups`	string array	Physical frequency groups: `hf`,

		`vhf`, `uhf`, `shf`.

9.2 Connection Status Values

Current wire values:

Value	Meaning
`unknown`	State not known yet.
`disconnected`	Radio is not connected.
`connecting`	Connection attempt in progress.
`reconnecting`	Reconnection attempt in progress after a failure.
`connected`	Radio is connected and responding.
`no_response`	Port opened but the radio did not respond to expected commands.
`stopped`	Radio handler was explicitly stopped.

9.3 VFO State

Each `vfos` entry has this shape:

```
{
  "vfo": "A",
  "frequency": 14074000,
  "mode": "USB",
  "filterWidth": 2400,
  "frequencyOffset": null
}
```

Frequencies are always in Hz. If `frequencyOffset` is active for this VFO, this value is already the transformed public frequency; clients should still send the same kind of value back in `setFrequency`.

`frequencyOffset` is optional VFO-specific metadata. When present, it has the same VFO entry shape used in `RadioConfiguration.frequencyOffset.vfos[*]`: `enabled`, optional `label`, `offsetHz`, `direction`, optional `minFrequencyHz`, and optional `maxFrequencyHz`. Clients use it for labels and UI limits only; they must never apply it again to `frequency`.

9.4 RIT And XIT State

```
{
  "enabled": true,
  "offset": 120
}
```

`offset` is in Hz and may be positive or negative.

9.5 Metering State

```
{
  "sMeter": 92,
  "signalDbm": -80.8,
  "swr": 1.2,
  "power": 80,
  "alc": 0
}
```

Field	Type	Meaning
`sMeter`	int	Legacy normalized meter value. Its scale depends on the radio/handbook; retain it for compatibility, but do not interpret it as a literal S-unit number.
`signalDbm`	number or `null`	Received signal strength in dBm when the radio supplies an absolute reading. TCI radios populate this from messages such as `rx_smeter`, `rx_sensors`, or compatible extended sensor messages.
`swr`	number or `null`	Standing-wave ratio, normally meaningful while transmitting.
`power`	int	Transmit forward power reading, normally in watts when supplied by TCI.
`alc`	int	Automatic level control reading on radios that expose it.

Some values may be zero or `null` depending on the radio and handbook. For new integrations, prefer `signalDbm` when it is present and use `sMeter` only as a fallback.

Traditional S-unit presentation

`signalDbm` is the authoritative wire value. A client may convert it to the traditional HF S-meter presentation using the conventional ITU scale:

- S1 = -121 dBm;
- S9 = -73 dBm;
- each S-unit below S9 represents 6 dB;
- values above S9 are commonly displayed in 10 dB steps, such as `S9+10` or `S9+40`.

Examples:

`signalDbm`	Display
`-121`	`S1`
`-103`	`S4`
`-73`	`S9`
`-33`	`S9+40`

CAT4OM's own UI displays both forms, for example `S8 (-80.8 dBm)`. Third-party clients should derive presentation text locally rather than treating a formatted S-unit string as protocol state.

9.6 availableCommands

`availableCommands` tells clients which radio operations are declared by the active handbook. This is more precise than assuming every radio supports every operation.

Example:

```
"availableCommands": [
```

```
      "GetFrequencyA",
      "GetFrequencyB",
      "SetFrequency",
      "GetModeA",
      "SetMode",
      "SetVfo",
      "SetSplit",
      "SetPtt",
      "VfoAtoB",
      "VfoSwap"
    ]
```

Use it to hide or disable features:

Desired UI/API feature	Check
Frequency setting	`SetFrequency` exists.
Mode setting	`SetMode` exists.
Active VFO selection	`SetVfo` exists.
Split control	`SetSplit` exists.
PTT	`SetPtt` exists.
VFO A to B	`VfoAtoB` exists.
VFO B to A	`VfoBtoA` exists.
Native VFO equalize	`VfoEqualize` exists.
VFO swap	`VfoSwap` exists.
Power off	`PowerOff` exists.

Match command names case-insensitively in your client for user-interface gating, but send action names exactly as documented.

9.7 commandVfoScopes

`commandVfoScopes` explains non-default VFO targeting behavior.

Example:

```
  "commandVfoScopes": {
    "SetFrequency": "activeOnly"
  }
```

This means that the radio can set frequency only on the current active VFO. A client should not offer direct tuning for VFO B while VFO A is active. If the client ignores this and sends a request for the non-active VFO, the service rejects it with `INVALID_TARGET_VFO` and does not send a command to the radio.

Missing map entries mean default behavior: the command is assumed to be usable with the requested target.

10. Control Push Messages

10.1 stateUpdate

The server pushes `stateUpdate` when radio state changes. It also sends keepalive state updates approximately every 10 seconds while clients are connected.

```
  {
```

```
    "type": "stateUpdate",
    "timestamp": "2026-06-02T20:30:00Z",
    "masterId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
    "radios": [
      {
        "radioId": "run",
        "groupId": "main",
        "radioName": "Run Radio",
        "connectionStatus": "connected",
        "vfos": {
          "A": {
            "vfo": "A",
            "frequency": 14074500,
            "mode": "USB",
            "filterWidth": 2400
          }
        },
        "activeVfo": "A",
        "txVfo": "A",
        "rxVfos": ["A"],
        "split": false,
        "rit": {
          "enabled": false,
          "offset": 0
        },
        "xit": {
          "enabled": false,
          "offset": 0
        },
        "ptt": false,
        "metering": {
          "sMeter": 0,
          "signalDbm": null,
          "swr": null,
          "power": 0,
          "alc": 0
        },
        "supportedModes": ["USB", "LSB", "CW"],
        "availableCommands": ["SetFrequency", "SetMode", "SetPtt"],
        "commandVfoScopes": {},
        "frequencyResolution": 1,
        "frequencyGroups": ["hf"]
      }
    ]
  }
```


Clients should:

14. Replace their local radio state from each pushed snapshot.
15. Update local ownership by comparing `masterId` to their own `clientId`.
16. Keep reading messages even while waiting for command responses.

10.2 event

Events are server-to-client notifications that are not full radio state snapshots.

Common events:

Event	Meaning
`ownershipChanged`	The group master changed.
`diagnosticsCompleted`	A diagnostic run finished or was cancelled.

Ownership example:

```
{
  "type": "event",
  "event": "ownershipChanged",
  "timestamp": "2026-06-02T20:31:00Z",
  "radioId": null,
  "details": {
    "masterId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38"
  }
}
```

Diagnostics completion example:

```
{
  "type": "event",
  "event": "diagnosticsCompleted",
  "timestamp": "2026-06-02T20:35:00Z",
  "radioId": "run",
  "details": {
    "cancelled": false,
    "total": 24,
    "passed": 21,
    "failed": 1,
    "unverified": 1,
    "skipped": 1,
    "durationMs": 18342
  }
}
```

`details` fields: `cancelled` (bool), `total`, `passed`, `failed`, `unverified`, `skipped`, `durationMs`. `passed` are steps whose effect was read back and matched; `unverified` are steps whose command executed but the radio offered no way to read the result back (distinct from `skipped`, which means the command is not available on the radio); `failed` are mismatches or errors. Detailed per-step outcomes are not in the event — they are in the server's per-radio wire log.

Detailed per-step diagnostics results are written to the radio wire log, not returned over WebSocket.

11. Control Actions

11.1 Read Actions

Action	Purpose	`radioId`	Params	Result
`getAllState`	Returns all radio states in the group.	no	none	`{ "radios": [RadioState] }`
`getState`	Returns one radio state.	yes	none	`RadioState`
`getOwnership`	Makes the caller	no	none	`{ "role": "master" }`

	master.			

Get all state:

```
{
  "type": "request",
  "id": "state-all-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "getAllState",
  "params": {}
}
```

Get one radio state:

```
{
  "type": "request",
  "id": "state-run-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "getState",
  "radioId": "run",
  "params": {}
}
```

11.2 Write Actions

All write actions require master ownership.

Action	Purpose	Required params	Optional params
`setFrequency`	Sets frequency in Hz.	`frequency`	`vfo`
`setMode`	Sets operating mode.	`mode`	`vfo`
`setPtt`	Controls transmit state.	`enabled`	none
`setVfo`	Selects active VFO.	`vfo`	none
`setSplit`	Enables/disables split.	`enabled`	`txVfo`
`setRit`	Enables/disables RIT.	`enabled`	`offset`
`setXit`	Enables/disables XIT.	`enabled`	`offset`
`vfoAtoB`	Copies VFO A to VFO B when supported.	none	none
`vfoBtoA`	Copies VFO B to VFO A when supported.	none	none
`vfoEqualize`	Runs the radio-native VFO equalize operation when supported.	none	none
`vfoSwap`	Swaps VFO A and VFO B when supported.	none	none
`sendRaw`	Sends raw bytes to the radio.	`data`	none
`sendRawCmdFireAndForget`	Sends raw bytes to the radio without waiting for a radio response.	`data`	none
`sendCw`	Sends a CW text string via the radio's built-in keyer. Arms TX watchdog.	`text`	`wpm`

`stopCw`	Aborts the current CW transmission. Disarms TX watchdog.	none	none
`setCwSpeed`	Sets the CW keyer speed in WPM without transmitting.	`wpm`	none
`cwKeyDown`	Keys or unkeys the transmitter manually (tune).	`enabled`	none
`runDiagnostics`	Starts asynchronous diagnostics.	none	none
`cancelDiagnostics`	Cancels diagnostics for the radio.	none	none
`powerOff`	Powers off the radio via CAT. Only available when `PowerOff` is listed in `availableCommands`. After a successful command the server disconnects the radio immediately.	none	none
`reconnectDisconnectedRadios`	Retries connection on every runtime radio in the group that is currently disconnected or in adapter error. This action is group-level: omit `radioId`. Already connected or in-progress radios are skipped.	none	none

11.3 setFrequency

Frequency is expressed in Hz as an integer.

```
{
  "type": "request",
  "id": "freq-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setFrequency",
  "radioId": "run",
  "params": {
    "frequency": 14074000,
    "vfo": "A"
  }
}
```

If `vfo` is omitted, the active VFO is used.

Success:

```
{
  "type": "response",
  "id": "freq-1",
  "success": true,
```

```
  "radioId": "run",
  "result": null
}
```

The visible frequency should be updated from the next `stateUpdate`.

For VFOs with `vfos[*].frequencyOffset`, the request still uses the visible/public frequency. Example: if VFO `A` should make the physical radio tune `7,200,100` Hz and that VFO has an `add` offset of `10,000,000,000` Hz, the client sends `10,007,200,100` Hz with `vfo: "A"`. The server resolves the target VFO first, subtracts that VFO's offset before building the CAT command, and sends the real frequency to the radio. If the resulting physical radio frequency would be non-positive, or if the visible request is outside that VFO offset's `minFrequencyHz` / `maxFrequencyHz`, the server rejects the command with `FREQUENCY_OUT_OF_RANGE` before touching the radio.

11.4 setMode

```
{
  "type": "request",
  "id": "mode-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setMode",
  "radioId": "run",
  "params": {
    "mode": "USB",
    "vfo": "A"
  }
}
```

The server uppercases the mode before dispatching. Valid modes depend on `RadioState.supportedModes` and the active handbook. Common mode names include:

```
LSB, USB, CW, CW-R, AM, AM-N, FM, FM-N, RTTY, RTTY-R,
RTTY-LSB, RTTY-USB, DATA-USB, DATA-LSB, DATA-FM, FT8,
FT4, DV, C4FM
```

If `vfo` is omitted, the active VFO is used — same semantics as `setFrequency` (§11.3). Some radios can set mode only on the active VFO; the service publishes that limitation per-command in `commandVfoScopes` (`"SetMode": "activeOnly"`) and rejects requests for any other VFO with `INVALID_TARGET_VFO`, exactly like `setFrequency`. On radios whose handbook has no `setVfo` command in `availableCommands` (independently addressable VFOs, e.g. slice-based FlexRadio), `setMode` is the only way to change the mode of a non-active VFO, since there is no CAT command to switch which VFO is "active" — target it directly with `vfo` instead.

11.5 setPtt

```
{
  "type": "request",
  "id": "ptt-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setPtt",
  "radioId": "run",
  "params": {
    "enabled": true
  }
}
```

`enabled: true` means transmit. `enabled: false` means receive.

PTT should be handled carefully. A client should avoid leaving PTT enabled across disconnects, application shutdown, or exceptions. For safety, send `setPtt false` before closing if your application may have enabled transmit.

While the radio is transmitting, the TX safety lock (section 8.4) rejects every command except `setPtt false` and read-only queries with `TX_IN_PROGRESS`, and freezes `getOwnership`. As a backstop the server also enforces a TX watchdog that forces the radio back to RX after `txTimeoutSeconds` (default 120 s, max 180 s), so do not rely on it as a normal control path — always unkey explicitly.

Malformed `enabled` — failsafe behavior. Unlike other commands, a `setPtt` whose `enabled` parameter is missing or not a valid boolean is *not* rejected with `INVALID_PARAM`. The server fails safe: it forces PTT **off** and returns the unkey outcome with `status` set to `"FAILSAFE_UNKEY"`. Rationale: rejecting a garbled packet could leave a transmitter keyed when the client's intent was to stop transmitting. A client that receives `FAILSAFE_UNKEY` should treat its previous request as not honored and resend a well-formed command.

11.6 setVfo

```
{
  "type": "request",
  "id": "vfo-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setVfo",
  "radioId": "run",
  "params": {
    "vfo": "B"
  }
}
```

Allowed VFO values are normally taken from `RadioState.availableVfos`. Treat VFO names as opaque stable identifiers supplied by the handbook/runtime state: `A`, `B`, `C`, `MAIN`, `SUB`, or any other valid name a handbook defines. Do not assume fixed A/B properties; if the VFO is named `pippo`, clients should send and display `pippo`.

Some radios have independently addressable VFOs and no CAT concept of an "active VFO" to switch — their handbook has no `SetVfo` command, so `availableCommands` never contains `setVfo` (slice-based FlexRadio is the current example). Sending `setVfo` to one of these radios fails; check `availableCommands` first. `setFrequency`/`setMode` remain fully usable on any VFO for these radios via their own `vfo` parameter (§11.3, §11.4) — there is no need for an active-VFO switch to target VFO B.

11.7 setSplit

```
{
  "type": "request",
  "id": "split-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setSplit",
  "radioId": "run",
  "params": {
    "enabled": true,
    "txVfo": "B"
  }
}
```

`txVfo` is optional. When split is enabled, some radios infer the transmit VFO from their own state or from the active VFO. CAT4OM also performs optimistic state updates for some push-based protocols where the radio does not provide a separate pushed TX VFO message.

11.8 setRit And setXit

RIT:

```
{
  "type": "request",
  "id": "rit-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setRit",
  "radioId": "run",
  "params": {
    "enabled": true,
    "offset": 120
  }
}
```

XIT:

```
{
  "type": "request",
  "id": "xit-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setXit",
  "radioId": "run",
  "params": {
    "enabled": true,
    "offset": -80
  }
}
```

`offset` is optional and is expressed in Hz. When present it must be within ±9999 Hz; out-of-range values are rejected with `INVALID_PARAM` and never reach the radio.

11.9 VFO Operations

These operations are only meaningful when the radio handbook declares the matching command in `availableCommands`.

```
{
  "type": "request",
  "id": "swap-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "vfoSwap",
  "radioId": "run",
  "params": {}
}
```

The available operations are:

Action	Meaning
`vfoAtoB`	Copy VFO A into VFO B.
`vfoBtoA`	Copy VFO B into VFO A.
`vfoEqualize`	Execute the radio-native equalize operation. Its exact direction is defined by the radio protocol/handbook.

`vfoSwap`	Swap VFO A and VFO B.

11.10 sendRaw

`sendRaw` sends raw bytes to the radio adapter. The `data` parameter is Base64.

For ASCII command `FA;`, bytes are:

```
46 41 3B
```

Base64:

```
RkE7
```

Request:

```
{
  "type": "request",
  "id": "raw-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "sendRaw",
  "radioId": "run",
  "params": {
    "data": "RkE7"
  }
}
```

Do not send hexadecimal text in `data`. Encode the exact bytes as Base64.

`sendRaw` is powerful and bypasses high-level semantic validation. Use it only for diagnostics, vendor-specific commands, or integration experiments.

11.11 sendRawCmdFireAndForget

`sendRawCmdFireAndForget` sends raw bytes to the radio adapter and returns success after the adapter accepts the write. It does not wait for a radio response. The `data` parameter is Base64 and uses the same format as `sendRaw`.

Use this action for vendor-specific messages that are naturally one-way. The server does not interpret the payload and does not infer any radio state change from this write.

Clients should still gate usage by the target radio's published capabilities. For example, if an integration wants to send a one-way raw vendor message, it should first check that the target `RadioState.availableCommands` contains `SendRaw`, then apply any integration-specific protocol policy using the published radio metadata such as `handbookId`, `manufacturer`, and `model`. CAT4OM does not treat `sendRawCmdFireAndForget` as a TCI-specific action.

Request:

```
{
  "type": "request",
  "id": "raw-ff-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "sendRawCmdFireAndForget",
  "radioId": "run",
  "params": {
    "data": "RkE7"
  }
}
```

11.12 Diagnostics

Start diagnostics:

```
{
  "type": "request",
  "id": "diag-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "runDiagnostics",
  "radioId": "run",
  "params": {}
}
```

Immediate response:

```
{
  "type": "response",
  "id": "diag-1",
  "success": true,
  "status": "DIAGNOSTICS_STARTED"
}
```

The diagnostic run continues in the background. Completion is reported with a `diagnosticsCompleted` event.

Cancel diagnostics:

```
{
  "type": "request",
  "id": "diag-cancel-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "cancelDiagnostics",
  "radioId": "run",
  "params": {}
}
```

Diagnostics are master-only because they can temporarily suspend polling and run mutating round-trip tests.

11.12 sendCw

`sendCw` queues a CW text string for transmission via the radio's built-in hardware keyer. It is only available when the target radio's handbook defines the `CwSend` command (check `RadioState.availableCommands`).

`sendCw` and `cwKeyDown(enabled: true)` are the only actions that key the transmitter without going through `setPtt`. To ensure the TX safety lock covers the CW transmission, the server sets `ptt = true` on the `RadioState` and arms the TX watchdog before issuing the hardware command. The `ptt` field in subsequent `stateUpdate` messages will therefore be `true` for the duration of the CW text.

```
{
  "type": "request",
  "id": "cw-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "sendCw",
  "radioId": "run",
  "params": {
    "text": "CQ DX DE IU4APC K",
    "wpm": 25
  }
```

```
}
```

`wpm` is optional. If omitted, the radio's current keyer speed is used (or the last speed set via `setCwSpeed`).

The action is rejected with `CW_NOT_SUPPORTED` if the target radio does not expose `CwSend` in `availableCommands`.

11.13 stopCw

`stopCw` immediately aborts the current CW transmission and clears the keyer queue. It also disarms the TX watchdog and sets `ptt = false` on the `RadioState`. `stopCw` does not require the TX safety gate to be clear — it is always accepted while CW is in progress.

```
{
  "type": "request",
  "id": "cw-stop-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "stopCw",
  "radioId": "run",
  "params": {}
}
```

11.14 setCwSpeed

`setCwSpeed` sets the keyer speed in WPM without transmitting. The speed is applied immediately and persists for subsequent `sendCw` calls that omit the `wpm` parameter.

```
{
  "type": "request",
  "id": "cw-speed-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "setCwSpeed",
  "radioId": "run",
  "params": {
    "wpm": 22
  }
}
```

`setCwSpeed` does not key the transmitter and is not gated by the TX safety lock. `wpm` must be between 1 and 100; out-of-range values are rejected with `INVALID_PARAM` and never reach the radio.

11.15 cwKeyDown

`cwKeyDown` keys or unkeys the transmitter manually for tune or carrier purposes. Like `sendCw`, when `enabled` is `true` the server sets `ptt = true` on the `RadioState` and arms the TX watchdog before issuing the hardware command.

```
{
  "type": "request",
  "id": "cw-key-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "cwKeyDown",
  "radioId": "run",
  "params": {
    "enabled": true
  }
}
```

Send `enabled: false` to release the key. The client is responsible for sending the unkey within a reasonable time; the TX watchdog will force unkey after `txTimeoutSeconds` if the client fails to do so.

11.16 powerOff

`powerOff` sends the handbook `PowerOff` command to the radio via CAT, then immediately disconnects the adapter on the server side. The radio transitions to `disconnected` state within the same server operation rather than waiting for the serial link to go silent.

Check `availableCommands` before offering this action: it is only present when the radio's handbook defines a `PowerOff` command.

```
{
  "type": "request",
  "id": "pwroff-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "powerOff",
  "radioId": "run",
  "params": {}
}
```

Success (radio is powering down and the server has disconnected):

```
{
  "type": "response",
  "id": "pwroff-1",
  "success": true,
  "radioId": "run",
  "result": null
}
```

Shortly after the success response, the client receives a `stateUpdate` with `connectionStatus: "disconnected"`.

If the handbook does not define `PowerOff`, the server returns `NOT_SUPPORTED`. If the radio is not connected, the server returns `RADIO_NOT_CONNECTED`.

`powerOff` is also available on the Management Port as `powerOffRadio` (see §6.13).

11.17 reconnectDisconnectedRadios

`reconnectDisconnectedRadios` is a group-level Control Port action. It tells the server to make one immediate connection attempt for every runtime radio in the connected group whose adapter state is `disconnected` or `error`.

The action requires master ownership. Do not send `radioId`; the server evaluates the whole group. Radios that are already `connected`, `connecting`, or `reconnecting` are reported as skipped.

```
{
  "type": "request",
  "id": "reconnect-all-1",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "action": "reconnectDisconnectedRadios",
  "params": {}
}
```

Success:

```
{
  "type": "response",
```

```
  "id": "reconnect-all-1",
  "success": true,
  "result": {
    "groupId": "default",
    "attempted": 1,
    "reconnected": 1,
    "failed": 0,
    "skipped": 2,
    "radios": [
      {
        "radioId": "main",
        "attempted": true,
        "before": "disconnected",
        "after": "connected",
        "connected": true,
        "message": "Reconnect succeeded."
      }
    ]
  }
}
```

The normal `stateUpdate` stream remains authoritative after the response.

12. Error Handling

Every client should follow these rules:

17. Always check `success`.
18. If `success` is `false`, inspect `error.code`.
19. Treat `error.message` as human-readable diagnostics, not as a stable machine API.
20. For `setConfiguration`, also inspect `errors[]` when present.
21. Keep receiving push messages while waiting for command responses.
22. Time out pending requests on the client side. Ten seconds is a reasonable default for most management/control commands, but radio operations may occasionally take longer depending on transport and timeout settings.

Common error codes:

Code	Meaning
`INTERNAL_ERROR`	Unexpected server-side failure.
`INVALID_REQUEST`	Malformed JSON or invalid request shape.
`MISSING_PARAM`	Required parameter missing.
`INVALID_PARAM`	Parameter missing or invalid. Required Control parameters are validated strictly: `setFrequency` rejects a missing/non-integer `frequency`; `setSplit`/`setRit`/`setXit` reject a missing/non-boolean `enabled` and a present-but-non-integer `offset`; `setRit`/`setXit` also reject an `offset` outside ±9999 Hz; `setCwSpeed` rejects a `wpm` outside 1-100; `sendRaw`/`sendRawCmdFireAndForget` reject a missing or non-Base64 `data`. Exception: `setPtt` never rejects a malformed `enabled` — it fails safe to PTT off (see §11.5, `FAILSAFE_UNKEY`).

`UNKNOWN_ACTION`	Action name not recognized.
`NOT_FOUND`	Requested group/handbook/resource not found.
`NOT_READY`	Runtime component not initialized or group not ready.
`NOT_SUPPORTED`	Feature not supported by the radio/handbook.
`NOT_MASTER`	Client attempted a master-only action while slave.
`RADIO_CLAIMED`	The target radio is under exclusive control of a legacy emulation session (Hamlib NET rigctl). Native clients cannot command it until that session disconnects. The radio's `legacyControlled` state field is `true`. See §21.
`NOT_CONNECTED`	Radio/transport not connected.
`TIMEOUT`	Radio/transport operation timed out.
`CANCELED`	Operation was cancelled.
`TRANSPORT_ERROR`	Adapter-level transport error.
`BUILD_FAILED`	CAT frame could not be built from handbook/params.
`SEND_FAILED`	CAT frame send failed.
`RADIO_REJECTED`	Radio rejected the command.
`PARSE_FAILED`	Radio response could not be parsed.
`STEP_REJECTED`	A command sequence step was rejected.
`COMMAND_FAILED`	Generic radio command failure.
`INVALID_MODE`	Requested mode missing or invalid.
`INVALID_VFO`	Requested VFO missing or invalid.
`INVALID_TARGET_VFO`	Command cannot target the requested VFO.
`INVALID_CONFIG`	Configuration payload invalid.
`EMPTY_GROUP_ID`	A group has an empty ID. Used by `setConfiguration`.
`DUPLICATE_GROUP_ID`	Two or more groups share the same ID. Used by `setConfiguration`.
`EMPTY_RADIO_ID`	A radio has an empty ID. Used by `setConfiguration`.
`DUPLICATE_RADIO_ID`	Two or more radios inside the same group share the same ID. Used by `setConfiguration`.
`MISSING_RADIO_HANDBOOK`	A radio has no handbook configured. Used by `setConfiguration`.
`MISSING_SERIAL_PORT`	A serial radio has no serial port configured. Used by `setConfiguration`.
`MGMT_PORT_CONFLICT`	A group Control Port conflicts with the Management Port. Used by `setConfiguration`.
`RECONNECT_INTERVAL_RANGE`	A group `reconnectIntervalSeconds` is outside `1..3600`. Used by `setConfiguration`.
`INVALID_FREQUENCY_OFFSET`	A configured radio frequency offset is enabled but invalid. Used by `setConfiguration`.

`INVALID_FREQUENCY_OFFSET_DIRECTION`	A configured radio frequency offset direction is not supported. Used by `setConfiguration`.
`INVALID_FREQUENCY_OFFSET_RANGE`	A configured radio frequency offset min/max range is invalid. Used by `setConfiguration`.
`GROUP_MUST_BE_STOPPED_FOR_CONFIGURATION_CHANGE`	The requested configuration changes group-level settings or radio membership for a running group. Used by `setConfiguration`.
`RADIO_MUST_BE_STOPPED_FOR_CONFIGURATION_CHANGE`	The requested configuration changes master configuration for an active radio. Used by `setConfiguration`.
`CONTROL_PORT_IN_USE`	A group Control Port is already bound by another running group. Logged during runtime start validation; Management callers receive `START_FAILED` or `RESTART_FAILED`.
`FREQUENCY_OUT_OF_RANGE`	A `setFrequency` request is outside the configured public frequency range for the target VFO's active `frequencyOffset`.
`TX_IN_PROGRESS`	The TX safety lock rejected the request: a radio is transmitting. Only `setPtt false` and read-only queries are accepted on a transmitting radio, and `getOwnership` is frozen while any radio transmits. See section 8.4.
`CW_NOT_SUPPORTED`	The target radio does not support CW keying (`CwSend` is absent from `availableCommands`). Returned by `sendCw`, `stopCw`, `setCwSpeed`, and `cwKeyDown`.
`INVALID_TX_TIMEOUT`	A radio `txTimeoutSeconds` is negative. Used by `setConfiguration`.
`TX_TIMEOUT_CAPPED`	Warning: a radio `txTimeoutSeconds` exceeds the maximum and will be capped to 180 s at load. Used by `setConfiguration`.
`SAVE_ERROR`	Configuration save failed.
`INVALID_LOG_LEVEL`	Requested log level not supported.
`LOG_LEVEL_SAVE_ERROR`	Log level could not be saved/applied.
`START_FAILED`	Group/radio start failed.
`STOP_FAILED`	Radio stop failed.
`RESTART_FAILED`	Group restart failed.
`CONNECTION_FAILED`	Test connection could not open serial port.
`TEST_FAILED`	Test connection failed with an exception.
`RADIO_NOT_FOUND`	Radio ID was not present in the running group.

13. Robust Client Behavior

13.1 Receive Loop

Do not write a client that sends a request and then blocks the whole WebSocket until the response arrives. Push messages may arrive at any time:

```
client sends setFrequency
```

```
server sends stateUpdate
server sends response for setFrequency
server sends another stateUpdate
```

Your client should have a receive loop that:

23. Reads every WebSocket text message.
24. Parses the `type`.
25. Routes `response` by `id` to a pending request.
26. Applies `stateUpdate` to local state.
27. Applies `event` to local ownership/diagnostics UI.

13.2 Reconnection

On disconnect:

28. Mark local connection as disconnected.
29. Clear pending requests.
30. Reconnect with backoff.
31. Send `hello` again as the first message.
32. Wait for the welcome again.
33. Refresh state from welcome or `getAllState`.
34. Do not assume you are still master.

Ownership is connection-scoped. If you disconnect and reconnect, you receive a new `clientId` and a new role.

13.3 State Authority

Treat pushed state as truth. A command response means CAT4OM accepted and executed the operation as far as the command pipeline knows. The radio state may still normalize differently. For example:

Request	Possible radio reality
Set frequency to `14074003`	Radio rounds to `14074000` because its resolution is 10 Hz or 100 Hz.
Set mode to `DATA-USB`	Radio reports `USB-D` or another mapped handbook value normalized by CAT4OM.
Enable split	TX VFO may be inferred differently depending on radio protocol.

Use `stateUpdate` to present the final state.

13.4 Feature Gating

Before showing or sending a command:

35. Check `connectionStatus == "connected"`.
36. Check you are master for write actions.
37. Check `availableCommands`.
38. Check `commandVfoScopes`.
39. Check `supportedModes` and `availableVfos`.

This prevents noisy errors and makes the integration feel native to each radio.

14. Complete C# Example Without CAT4OM Contracts

This example uses only `ClientWebSocket` and `System.Text.Json`. It connects to a Control Port, sends the `hello` handshake, requests ownership if needed, sets a frequency, keeps receiving push messages, and displays `signalDbm` using traditional S-units.

```
using System.Collections.Concurrent;
using System.Net.WebSockets;
using System.Text;
using System.Text.Json;

public sealed class Cat4OmJsonControlClient : IAsyncDisposable
{
    private readonly ClientWebSocket _ws = new();
    private readonly ConcurrentDictionary<string,
TaskCompletionSource<JsonElement>> _pending = new();
    private readonly CancellationTokenSource _cts = new();
    private string? _clientId;
    private string? _masterId;

    public async Task ConnectAsync(string host, int port)
    {
        await _ws.ConnectAsync(new Uri($"ws://{host}:{port}/"), _cts.Token);

        // Single-step handshake: send 'hello' as the first message, then read the
reply.
        var hello = JsonSerializer.Serialize(new
        {
            type = "hello",
            protocolVersion = "1.0.0",
            appName = "ExampleCSharpClient",
            appVersion = "1.0.0"
        });
        await _ws.SendAsync(Encoding.UTF8.GetBytes(hello),
WebSocketMessageType.Text, true, _cts.Token);

        var reply = JsonDocument.Parse(await
ReceiveTextAsync()).RootElement.Clone();
        var replyType = reply.GetProperty("type").GetString();
        if (replyType == "error")
            throw new InvalidOperationException(
                $"Connection rejected: [{reply.GetProperty("code").GetString()}]
{reply.GetProperty("message").GetString()}");

        RequireString(reply, "type", "welcome");
        RequireString(reply, "endpoint", "control");

        var protocolVersion = reply.GetProperty("protocolVersion").GetString();
        if (protocolVersion != "1.0.0")
            throw new InvalidOperationException($"Unsupported CAT4OM protocol:
{protocolVersion}");

        _clientId = reply.GetProperty("clientId").GetString();
```

```
        if (reply.GetProperty("role").GetString() == "master")
            _masterId = _clientId;
        PrintMetering(reply);

        // Ready to send commands as soon as the welcome arrives.
        _ = Task.Run(ReceiveLoopAsync);
    }

    public async Task EnsureMasterAsync()
    {
        if (_clientId == _masterId)
            return;

        await SendRequestAsync("getOwnership", null, new { });
        _masterId = _clientId;
    }

    public async Task SetFrequencyAsync(string radioId, long frequencyHz, string?
vfo = null)
    {
        await EnsureMasterAsync();

        object parameters = vfo is null
            ? new { frequency = frequencyHz }
            : new { frequency = frequencyHz, vfo };

        await SendRequestAsync("setFrequency", radioId, parameters);
    }

    private async Task<JsonElement> SendRequestAsync(string action, string?
radioId, object? parameters)
    {
        if (_clientId is null)
            throw new InvalidOperationException("Not connected.");

        var id = Guid.NewGuid().ToString("N");
        var tcs = new
TaskCompletionSource<JsonElement>(TaskCreationOptions.RunContinuationsAsynchronous
ly);
        _pending[id] = tcs;

        var request = new
        {
            type = "request",
            id,
            clientId = _clientId,
            action,
            radioId,
            @params = parameters
        };

        var json = JsonSerializer.Serialize(request);
```

```
        var bytes = Encoding.UTF8.GetBytes(json);
        await _ws.SendAsync(bytes, WebSocketMessageType.Text, true, _cts.Token);

        using var timeout = new CancellationTokenSource(TimeSpan.FromSeconds(10));
        using var registration = timeout.Token.Register(() =>
tcs.TrySetCanceled());

        var response = await tcs.Task;
        if (!response.GetProperty("success").GetBoolean())
        {
            var error = response.TryGetProperty("error", out var e) ? e : default;
            var code = error.ValueKind == JsonValueKind.Object &&
error.TryGetProperty("code", out var c)
                ? c.GetString()
                : "UNKNOWN";
            var message = error.ValueKind == JsonValueKind.Object &&
error.TryGetProperty("message", out var m)
                ? m.GetString()
                : "Command failed";
            throw new InvalidOperationException($"{code}: {message}");
        }

        return response;
    }

    private async Task ReceiveLoopAsync()
    {
        while (_ws.State == WebSocketState.Open && !_cts.IsCancellationRequested)
        {
            var json = await ReceiveTextAsync();
            var message = JsonDocument.Parse(json).RootElement.Clone();
            var type = message.GetProperty("type").GetString();

            if (type == "response")
            {
                var id = message.GetProperty("id").GetString();
                if (id is not null && _pending.TryRemove(id, out var tcs))
                    tcs.TrySetResult(message);
            }
            else if (type == "stateUpdate")
            {
                _masterId = message.TryGetProperty("masterId", out var master)
                    ? master.GetString()
                    : null;

                PrintMetering(message);
                Console.WriteLine("State update received.");
            }
            else if (type == "event")
            {
                var eventName = message.GetProperty("event").GetString();
                if (eventName == "ownershipChanged" &&
```

```
                    message.TryGetProperty("details", out var details) &&
                    details.TryGetProperty("masterId", out var master))
                {
                    _masterId = master.GetString();
                }
            }
        }
    }

    private async Task<string> ReceiveTextAsync()
    {
        var buffer = new byte[16 * 1024];
        using var ms = new MemoryStream();

        while (true)
        {
            var result = await _ws.ReceiveAsync(buffer, _cts.Token);
            if (result.MessageType == WebSocketMessageType.Close)
                throw new WebSocketException("WebSocket closed by server.");

            ms.Write(buffer, 0, result.Count);
            if (result.EndOfMessage)
                return Encoding.UTF8.GetString(ms.ToArray());
        }
    }

    private static void RequireString(JsonElement element, string property, string
expected)
    {
        var actual = element.GetProperty(property).GetString();
        if (actual != expected)
            throw new InvalidOperationException($"Expected {property}={expected},
received {actual}.");
    }

    private static void PrintMetering(JsonElement message)
    {
        if (!message.TryGetProperty("radios", out var radios) || radios.ValueKind
!= JsonValueKind.Array)
            return;

        foreach (var radio in radios.EnumerateArray())
        {
            if (!radio.TryGetProperty("metering", out var metering) ||
                !metering.TryGetProperty("signalDbm", out var signalDbm) ||
                signalDbm.ValueKind != JsonValueKind.Number)
                continue;

            var dbm = signalDbm.GetDouble();
            Console.WriteLine($"{radio.GetProperty("radioId").GetString()}:
{ToSUnit(dbm)} ({dbm:0.0} dBm)");
        }
```

```
    }

    private static string ToSUnit(double dbm)
    {
        if (dbm > -73)
        {
            var overS9 = (int)Math.Round((dbm + 73) / 10,
MidpointRounding.AwayFromZero) * 10;
            return overS9 > 0 ? $"S9+{overS9}" : "S9";
        }

        var sUnit = (int)Math.Round(9 + ((dbm + 73) / 6),
MidpointRounding.AwayFromZero);
        return $"S{Math.Clamp(sUnit, 0, 9)}";
    }

    public async ValueTask DisposeAsync()
    {
        _cts.Cancel();
        if (_ws.State == WebSocketState.Open)
            await _ws.CloseAsync(WebSocketCloseStatus.NormalClosure, "Closing",
CancellationToken.None);
        _ws.Dispose();
        _cts.Dispose();
    }
}
```

Usage:

```
await using var client = new Cat4OmJsonControlClient();
await client.ConnectAsync("localhost", 5001);
await client.SetFrequencyAsync("run", 14074000, "A");
```

15. Complete Java Example Without CAT4OM Contracts

This example uses the JDK `java.net.http.WebSocket` API and Jackson for JSON. It shows the same pattern: connect, send `hello`, receive the welcome, request ownership, send a command, handle pushes, and display `signalDbm` using traditional S-units.

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;

import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.WebSocket;
import java.util.Map;
import java.util.Locale;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CompletionStage;
import java.util.concurrent.TimeUnit;

public final class Cat4OmJsonControlClient implements WebSocket.Listener {
```

```
    private final ObjectMapper json = new ObjectMapper();
    private final Map<String, CompletableFuture<JsonNode>> pending = new
ConcurrentHashMap<>();
    // welcomeFuture is completed by onText with the first server message (welcome
or error).
    // Blocking calls (getOwnership, …) run outside the listener to avoid blocking
it.
    private final CompletableFuture<JsonNode> welcomeFuture = new
CompletableFuture<>();
    private final StringBuilder incoming = new StringBuilder();
    private WebSocket webSocket;
    private String clientId;
    private String masterId;

    /**
     * Opens the WebSocket, sends the hello as the first message, then waits for
the welcome
     * (or error) and validates it — all without blocking the WebSocket listener
thread.
     */
    public void connect(String host, int port) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        webSocket = client.newWebSocketBuilder()
                .buildAsync(URI.create("ws://" + host + ":" + port + "/"), this)
                .join();

        // Single-step handshake: send 'hello' as the first message.
        ObjectNode hello = json.createObjectNode();
        hello.put("type", "hello");
        hello.put("protocolVersion", "1.0.0");
        hello.put("appName", "ExampleJavaClient");
        hello.put("appVersion", "1.0.0");
        webSocket.sendText(json.writeValueAsString(hello), true);

        // Block the calling thread (not the listener) until the welcome (or
error) arrives.
        JsonNode reply = welcomeFuture.get(10, TimeUnit.SECONDS);

        if ("error".equals(reply.path("type").asText())) {
            throw new IllegalStateException("Connection rejected: ["
                    + reply.path("code").asText() + "] " +
reply.path("message").asText());
        }

        require(reply, "type", "welcome");
        require(reply, "endpoint", "control");

        String protocolVersion = reply.get("protocolVersion").asText();
        if (!protocolVersion.equals("1.0.0")) {
            throw new IllegalStateException("Unsupported CAT4OM protocol: " +
protocolVersion);
        }
```

```
        clientId = reply.get("clientId").asText();
        if ("master".equals(reply.get("role").asText())) {
            masterId = clientId;
        }
        printMetering(reply);
        // Ready to send commands as soon as the welcome arrives.
    }

    public void ensureMaster() throws Exception {
        if (clientId != null && clientId.equals(masterId)) {
            return;
        }
        sendRequest("getOwnership", null, json.createObjectNode()).get(10,
TimeUnit.SECONDS);
        masterId = clientId;
    }

    public void setFrequency(String radioId, long frequencyHz, String vfo) throws
Exception {
        ensureMaster();

        ObjectNode params = json.createObjectNode();
        params.put("frequency", frequencyHz);
        if (vfo != null && !vfo.isBlank()) {
            params.put("vfo", vfo);
        }

        sendRequest("setFrequency", radioId, params).get(10, TimeUnit.SECONDS);
    }

    private CompletableFuture<JsonNode> sendRequest(String action, String radioId,
JsonNode params)
            throws Exception {
        String id = UUID.randomUUID().toString();
        CompletableFuture<JsonNode> future = new CompletableFuture<>();
        pending.put(id, future);

        ObjectNode request = json.createObjectNode();
        request.put("type", "request");
        request.put("id", id);
        request.put("clientId", clientId);
        request.put("action", action);
        if (radioId != null) {
            request.put("radioId", radioId);
        }
        request.set("params", params);

        webSocket.sendText(json.writeValueAsString(request), true);
        return future.thenApply(response -> {
            if (!response.get("success").asBoolean()) {
                JsonNode error = response.get("error");
```

```
                String code = error != null && error.has("code") ?
error.get("code").asText() : "UNKNOWN";
                String msg = error != null && error.has("message") ?
error.get("message").asText() : "Command failed";
                throw new RuntimeException(code + ": " + msg);
            }
            return response;
        });
    }

    @Override
    public CompletionStage<?> onText(WebSocket webSocket, CharSequence data,
boolean last) {
        incoming.append(data);
        if (!last) {
            webSocket.request(1);
            return CompletableFuture.completedFuture(null);
        }

        try {
            JsonNode message = json.readTree(incoming.toString());
            incoming.setLength(0);

            String type = message.get("type").asText();
            switch (type) {
                case "welcome" -> {
                    // Complete the future so connect() can proceed; do NOT block
here.
                    welcomeFuture.complete(message);
                }
                case "error" -> {
                    // Connection-level rejection of the hello; surface it to
connect().
                    welcomeFuture.complete(message);
                }
                case "response" -> {
                    String id = message.get("id").asText();
                    CompletableFuture<JsonNode> future = pending.remove(id);
                    if (future != null) {
                        future.complete(message);
                    }
                }
                case "stateUpdate" -> {
                    JsonNode master = message.get("masterId");
                    masterId = master == null || master.isNull() ? null :
master.asText();
                    printMetering(message);
                    System.out.println("State update received");
                }
                case "event" -> {
                    if ("ownershipChanged".equals(message.get("event").asText()))
{
                        JsonNode details = message.get("details");
```

```
                        if (details != null && details.has("masterId")) {
                            masterId = details.get("masterId").asText();
                        }
                    }
                }
                default -> System.out.println("Unhandled message type: " + type);
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            welcomeFuture.completeExceptionally(ex);
        }

        webSocket.request(1);
        return CompletableFuture.completedFuture(null);
    }

    @Override
    public void onOpen(WebSocket webSocket) {
        webSocket.request(1);
    }

    private static void require(JsonNode node, String field, String expected) {
        String actual = node.get(field).asText();
        if (!actual.equals(expected)) {
            throw new IllegalStateException("Expected " + field + "=" + expected +
", received " + actual);
        }
    }

    private static void printMetering(JsonNode message) {
        JsonNode radios = message.get("radios");
        if (radios == null || !radios.isArray()) {
            return;
        }

        for (JsonNode radio : radios) {
            JsonNode dbmNode = radio.path("metering").get("signalDbm");
            if (dbmNode == null || !dbmNode.isNumber()) {
                continue;
            }

            double dbm = dbmNode.asDouble();
            System.out.printf(Locale.ROOT, "%s: %s (%.1f dBm)%n",
                    radio.path("radioId").asText(), toSUnit(dbm), dbm);
        }
    }

    private static String toSUnit(double dbm) {
        if (dbm > -73) {
            int overS9 = roundAwayFromZero((dbm + 73) / 10) * 10;
            return overS9 > 0 ? "S9+" + overS9 : "S9";
        }
```

```
            int sUnit = roundAwayFromZero(9 + ((dbm + 73) / 6));
            return "S" + Math.max(0, Math.min(9, sUnit));
        }

        private static int roundAwayFromZero(double value) {
            return value >= 0 ? (int) Math.floor(value + 0.5) : (int) Math.ceil(value
    - 0.5);
        }
    }
```

The key difference from a naive implementation: the first server message (welcome or error) is delivered to `connect()` via `welcomeFuture`, which is completed inside `onText`. This means all blocking calls happen on the calling thread, not inside the listener. The listener stays free to deliver responses at all times.

*The same client pattern in other languages: **Python** (§22) and **JavaScript / browser** (§23).*

16. Suggested Integration Flows

16.1 Read-Only Monitor

```
connect to Management Port
send hello with noRegister=true, receive managementWelcome
receive serviceStatus push
display groups, radios, connected clients
optionally connect to one Control Port
send hello with noRegister=true, receive welcome role=observer
display welcome.radios[]
apply stateUpdate.radios[]
never request ownership
never send write actions
```

16.2 Logging Program That Controls One Radio

```
know or discover group Control Port
connect to Control Port
send hello (as the logging program), receive welcome
read welcome.radios[]
find target radioId
if role is slave, send getOwnership
listen for ownershipChanged and stateUpdate
send setFrequency/setMode as needed
use stateUpdate as authoritative state
on shutdown, if PTT may be active, send setPtt false
close WebSocket
```

16.3 Configuration Tool

```
connect to Management Port
send hello, receive managementWelcome
getHandbooks
getAvailablePorts
getConfiguration
edit complete configuration locally
setConfiguration with full configuration object
restart affected groups
subscribe or getServiceStatus to verify runtime status
```

16.4 Diagnostics Tool

```
connect to Management Port
setLogLevel debug or trace if low-level logs are needed
connect to Control Port
send hello, receive welcome
getOwnership
runDiagnostics for selected radio
wait for diagnosticsCompleted event
read radio wire log from the server host if available
```

17. Security And Deployment Notes

Current CAT4OM builds expose fields such as `managementPassword`, group `authentication`, and `authRequired`, but authentication enforcement is reserved for future work. Treat the WebSocket endpoints as unauthenticated unless the deployment has external protection.

Recommended deployment practices:

- Use CAT4OM on trusted LANs.
- Restrict inbound firewall rules to known clients where possible.
- Avoid exposing Management or Control ports directly to the public internet.
- If remote internet access is required, place CAT4OM behind a VPN or an authenticated reverse proxy.
- Remember that Control Port master clients can command PTT and frequency changes.

18. Implementation Checklist

Use this checklist before calling an integration complete.

Item	Done
Send `hello` as the first message on both Management and Control ports.	
Validate `type`, `endpoint`, and `protocolVersion` on the welcome; handle an `error` reply.	
Use `hello.noRegister = true` only for read-only Management/Control observers that must not count as clients or participate in ownership.	
Keep a receive loop active at all times.	
Correlate responses by `id`.	
Handle `serviceStatus`, `stateUpdate`, and `event` messages that arrive between responses.	
Treat `success` as the authoritative response outcome.	
Branch on `error.code`, not on `error.message`.	
Use pushed state as authoritative after write commands.	
Implement reconnect with backoff.	
Do not assume ownership survives reconnect.	
Request ownership before master-only actions.	
React to `ownershipChanged`.	
Gate UI/features using `availableCommands`, `supportedModes`, `availableVfos`, and	

`commandVfoScopes`.	
Use Hz for all frequency values.	
Use Base64, not hex strings, for `sendRaw` and `sendRawCmdFireAndForget`.	
Send `setPtt false` during safe shutdown if your client may have enabled transmit.	
For configuration edits, send the complete configuration object, not a partial patch.	
If `setConfiguration` fails, inspect both `error` and `errors[]`.	

19. Quick Reference

19.1 Ports

Port	Default	URL
Management	`5000`	`ws://host:5000/`
Control	`5001+`	`ws://host:<groupControlPort>/`

19.2 Welcome Types

Channel	Welcome type	Endpoint
Management	`managementWelcome`	`management`
Control	`welcome`	`control`

19.3 Push Types

Channel	Push type	Meaning
Management	`serviceStatus`	Server/group/radio/client status.
Control	`stateUpdate`	Full radio state snapshot for the group.
Control	`event`	Ownership and diagnostics events.

19.4 Handshake (first message and reply)

Client sends `hello`:

```
{
  "type": "hello",
  "protocolVersion": "1.0.0",
  "appName": "YourApp",
  "appVersion": "1.0.0"
}
```

Add `"noRegister": true` only for a read-only observer that must not be counted as a client.

Server replies with a welcome on success (Control shown):

```
{
```

```
  "type": "welcome",
  "endpoint": "control",
  "protocolVersion": "1.0.0",
  "groupId": "main",
  "clientId": "f5f02de4-7d1c-40a1-9de0-8ea536b6ea38",
  "role": "slave",
  "sessionToken": null,
  "radios": [],
  "authRequired": false
}
```

…or with an `error` on rejection (then closes the connection):

```
{
  "type": "error",
  "code": "PROTOCOL_VERSION_MISMATCH",
  "message": "Unsupported protocol version '0.9', expected '1.0.0'."
}
```

19.5 Most Common Radio Command

```
{
  "type": "request",
  "id": "set-frequency",
  "clientId": "server-assigned-client-id",
  "action": "setFrequency",
  "radioId": "run",
  "params": {
    "frequency": 14074000,
    "vfo": "A"
  }
}
```

20. Source Alignment Notes

This manual is aligned with the current CAT4OM source layout:

Source area	Used for
`CAT4OM.Contracts.Protocol`	Message field names and protocol constants.
`CAT4OM.Contracts.Protocol.Management`	Management welcome, service status, group/radio/client status.
`CAT4OM.Contracts.Protocol.Control`	Control welcome, state updates, events.
`CAT4OM.Contracts.Radio`	`RadioState`, VFO state, metering state, connection status, mode names.
`CAT4OM.Contracts.Configuration`	Server configuration JSON.
`CAT4OM.Core.Services.ManagementServer`	Management action names, required params, result payloads, hello handshake behavior.
`CAT4OM.Core.Hub.RadioGroup`	Control action names, ownership behavior, master-only checks, diagnostics events.

When in doubt, the running server's JSON messages are authoritative for integration behavior.

21. Alternative Integration: Hamlib NET rigctl Command Subset

If your software already speaks the **Hamlib NET rigctl** protocol (the rig type used by WSJT-X, JTDX, fldigi, and many loggers), you may not need this manual at all. CAT4OM can expose **one radio of a group** over a dedicated TCP port through an independent emulation of a documented rigctl command subset. This is not a complete Hamlib or `rigctld` implementation. Point compatible software at the configured host and port (conventionally `4532`) using rig type *Hamlib NET rigctl*.

This path is intended for **existing programs** that cannot adopt the native WebSocket protocol. Choose between the two integration styles as follows:

	Native WebSocket protocol (this manual)	**Hamlib NET rigctl command subset**
Audience	New integrations, multi-radio control, full state/ownership model	Existing Hamlib-capable software
Transport	JSON over WebSocket (Management + Control ports)	rigctl text over TCP (one port per exposed radio)
Scope	All radios in a group, push state, ownership	One exposed radio, request/response
Concurrency	Many clients, master/slave arbitration	One controller at a time, exclusive

While a Hamlib client is connected, it holds the exposed radio exclusively; native clients see that radio's state read-only but keep full control of the other radios in the group. Enabling and behavior are described in the User Manual (§15, "Exposing a Radio to Hamlib / WSJT-X") and the Server Manual (§11, "Legacy Compatibility Bridge"). CAT4OM independently re-implements only its documented command

subset, does not include Hamlib software, and is not affiliated with the Hamlib project.

22. Complete Python Example

Same pattern as the C# and Java examples, using `asyncio` and the `websockets` package (`pip install websockets`): connect, send `hello`, read the welcome (or error), then keep a receive loop, correlate responses by `id`, and display `signalDbm` using traditional S-units.

```
import asyncio
import json
import math
import uuid
import websockets


class Cat4OmControlClient:
    def __init__(self):
        self._ws = None
        self._client_id = None
        self._master_id = None
        self._pending = {}  # request id -> asyncio.Future
```

```
    async def connect(self, host, port):
        self._ws = await websockets.connect(f"ws://{host}:{port}/")

        # Single-step handshake: send 'hello' as the first message, then read the
reply.
        await self._ws.send(json.dumps({
            "type": "hello",
            "protocolVersion": "1.0.0",
            "appName": "ExamplePythonClient",
            "appVersion": "1.0.0",
        }))

        reply = json.loads(await self._ws.recv())
        if reply.get("type") == "error":
            raise RuntimeError(f"Connection rejected: [{reply.get('code')}]
{reply.get('message')}")
        if reply.get("type") != "welcome" or reply.get("endpoint") != "control":
            raise RuntimeError(f"Unexpected handshake reply: {reply.get('type')}")
        if reply.get("protocolVersion") != "1.0.0":
            raise RuntimeError(f"Unsupported protocol:
{reply.get('protocolVersion')}")

        self._client_id = reply["clientId"]
        if reply.get("role") == "master":
            self._master_id = self._client_id
        print_metering(reply)

        # Ready to send commands as soon as the welcome arrives.
        asyncio.create_task(self._receive_loop())

    async def _receive_loop(self):
        async for raw in self._ws:
            msg = json.loads(raw)
            kind = msg.get("type")
            if kind == "response":
                future = self._pending.pop(msg.get("id"), None)
                if future and not future.done():
                    future.set_result(msg)
            elif kind == "stateUpdate":
                self._master_id = msg.get("masterId")
                print_metering(msg)
                print("State update received")
            elif kind == "event" and msg.get("event") == "ownershipChanged":
                self._master_id = (msg.get("details") or {}).get("masterId")

    async def _send_request(self, action, radio_id=None, params=None):
        req_id = uuid.uuid4().hex
        future = asyncio.get_running_loop().create_future()
        self._pending[req_id] = future

        request = {"type": "request", "id": req_id, "clientId": self._client_id,
"action": action}
```

```
        if radio_id is not None:
            request["radioId"] = radio_id
        if params is not None:
            request["params"] = params

        await self._ws.send(json.dumps(request))
        response = await asyncio.wait_for(future, timeout=10)
        if not response.get("success"):
            err = response.get("error") or {}
            raise RuntimeError(f"{err.get('code', 'UNKNOWN')}: {err.get('message',
'Command failed')}")
        return response

    async def ensure_master(self):
        if self._client_id == self._master_id:
            return
        await self._send_request("getOwnership")
        self._master_id = self._client_id

    async def set_frequency(self, radio_id, frequency_hz, vfo=None):
        await self.ensure_master()
        params = {"frequency": frequency_hz}
        if vfo:
            params["vfo"] = vfo
        await self._send_request("setFrequency", radio_id, params)


def round_away_from_zero(value):
    return math.floor(value + 0.5) if value >= 0 else math.ceil(value - 0.5)


def to_s_unit(dbm):
    if dbm > -73:
        over_s9 = round_away_from_zero((dbm + 73) / 10) * 10
        return f"S9+{over_s9}" if over_s9 > 0 else "S9"

    s_unit = round_away_from_zero(9 + ((dbm + 73) / 6))
    return f"S{max(0, min(9, s_unit))}"


def print_metering(message):
    for radio in message.get("radios", []):
        dbm = (radio.get("metering") or {}).get("signalDbm")
        if isinstance(dbm, (int, float)):
            print(f"{radio.get('radioId')}: {to_s_unit(dbm)} ({dbm:.1f} dBm)")


async def main():
    client = Cat4OmControlClient()
    await client.connect("localhost", 5001)
    await client.set_frequency("rig-1", 14_074_000)
```

```
    await asyncio.sleep(60)  # keep the connection open to receive pushed state


if __name__ == "__main__":
    asyncio.run(main())
```

23. Complete JavaScript (Browser) Example

For web dashboards, using the browser `WebSocket` API. The example also displays `signalDbm` using traditional S-units. The same pattern works in Node.js with the `ws` package (replace `new WebSocket(url)` with `new (require("ws"))(url)` and `crypto.randomUUID()` with a UUID helper).

```
class Cat4OmControlClient {
  constructor() {
    this.ws = null;
    this.clientId = null;
    this.masterId = null;
    this.pending = new Map(); // request id -> { resolve, reject }
  }

  connect(host, port) {
    return new Promise((resolve, reject) => {
      this.ws = new WebSocket(`ws://${host}:${port}/`);

      this.ws.addEventListener("open", () => {
        // Single-step handshake: send 'hello' as the first message.
        this.ws.send(JSON.stringify({
          type: "hello",
          protocolVersion: "1.0.0",
          appName: "ExampleBrowserClient",
          appVersion: "1.0.0",
        }));
      });

      this.ws.addEventListener("message", (event) => {
        const msg = JSON.parse(event.data);
        switch (msg.type) {
          case "welcome":
            if (msg.endpoint !== "control" || msg.protocolVersion !== "1.0.0") {
              reject(new Error(`Unexpected welcome:
${msg.endpoint}/${msg.protocolVersion}`));
              return;
            }
            this.clientId = msg.clientId;
            if (msg.role === "master") this.masterId = this.clientId;
            printMetering(msg);
            resolve(); // handshake complete; ready to send commands
            break;
          case "error":
            reject(new Error(`Connection rejected: [${msg.code}]
${msg.message}`));
            break;
          case "response": {
```

```
            const p = this.pending.get(msg.id);
            if (p) { this.pending.delete(msg.id); p.resolve(msg); }
            break;
          }
          case "stateUpdate":
            this.masterId = msg.masterId ?? null;
            printMetering(msg);
            console.log("State update received");
            break;
          case "event":
            if (msg.event === "ownershipChanged") {
              this.masterId = msg.details?.masterId ?? null;
            }
            break;
        }
      });

      this.ws.addEventListener("close", () => {
        for (const p of this.pending.values()) p.reject(new Error("Connection
closed"));
        this.pending.clear();
      });
      this.ws.addEventListener("error", () => reject(new Error("WebSocket
error")));
    });
  }

  sendRequest(action, radioId = null, params = null) {
    return new Promise((resolve, reject) => {
      const id = crypto.randomUUID();
      this.pending.set(id, {
        resolve: (msg) => {
          if (!msg.success) {
            const e = msg.error || {};
            reject(new Error(`${e.code || "UNKNOWN"}: ${e.message || "Command
failed"}`));
          } else {
            resolve(msg);
          }
        },
        reject,
      });

      const request = { type: "request", id, clientId: this.clientId, action };
      if (radioId !== null) request.radioId = radioId;
      if (params !== null) request.params = params;
      this.ws.send(JSON.stringify(request));
    });
  }

  async ensureMaster() {
    if (this.clientId === this.masterId) return;
```

```
    await this.sendRequest("getOwnership");
    this.masterId = this.clientId;
  }

  async setFrequency(radioId, frequencyHz, vfo = null) {
    await this.ensureMaster();
    const params = { frequency: frequencyHz };
    if (vfo) params.vfo = vfo;
    await this.sendRequest("setFrequency", radioId, params);
  }
}

function toSUnit(dbm) {
  if (dbm > -73) {
    const overS9 = roundAwayFromZero((dbm + 73) / 10) * 10;
    return overS9 > 0 ? `S9+${overS9}` : "S9";
  }

  const sUnit = roundAwayFromZero(9 + ((dbm + 73) / 6));
  return `S${Math.max(0, Math.min(9, sUnit))}`;
}

function roundAwayFromZero(value) {
  return value >= 0 ? Math.floor(value + 0.5) : Math.ceil(value - 0.5);
}

function printMetering(message) {
  for (const radio of message.radios ?? []) {
    const dbm = radio.metering?.signalDbm;
    if (typeof dbm === "number") {
      console.log(`${radio.radioId}: ${toSUnit(dbm)} (${dbm.toFixed(1)} dBm)`);
    }
  }
}

// Usage:
//   const client = new Cat4OmControlClient();
//   await client.connect("localhost", 5001);
//   await client.setFrequency("rig-1", 14074000);
```