Driving Firefox After CDP Removal: A WebDriver BiDi Adapter Case Study

Exzil Calanza

Independent Researcher, Iloilo City, Philippines

Independent research preprint. IEEE-style formatting only; this work has not been accepted by, published by, certified by, or peer reviewed by IEEE.

ABSTRACT This case study describes firefox-cdp, a lightweight Python adapter that maps a CDP-style tab interface onto Firefox's W3C WebDriver BiDi protocol over WebSockets after native CDP support was removed. The source article documents the protocol handshake, session initialization, remote-value deserialization, navigation and evaluation behavior, isolation boundaries, and self-test strategy. The implementation is an adapter library rather than a CDP network proxy: it offers a familiar Python-facing interface while speaking BiDi on the wire. The evidence is implementation-level and self-test based, not a cross-browser benchmark or proof of complete CDP compatibility.

INDEX TERMS ¢â‚¬â€□ browser automation; WebDriver BiDi; Firefox Remote Agent; Python WebSocket adapter; protocol transition; software tooling

I. INTRODUCTION

STUDY TYPE: Case study. **RESEARCH QUESTION:** How can a lightweight Python adapter expose a CDP-style tab interface over W3C WebDriver BiDi after Firefox removed native CDP support?

This preprint formalizes the existing public evidence record as a case study. No new experiment, dataset, measurement, or validation is claimed beyond the source article and its cited evidence.

If you want to drive a real Firefox from Python today, your options are a bit heavy: Selenium, geckodriver, or a Playwright server. That's fine when you want a full framework. It's a lot when all you want is a tiny object that can open a URL, run some JavaScript, take a screenshot, and type into a box — over a single websocket, the way people drive Chrome with a raw DevTools connection.

That lightweight path used to exist for Firefox too, through the Chrome DevTools Protocol. It doesn't anymore. So I wrote the ~700-line adapter that brings it back — **firefox-cdp** (MIT). One dependency (`websocket-client`), the rest is the standard library.

First, what it is and isn't. This is a **Python library you import** — not a network proxy and not a CDP server. It does **not** accept incoming CDP connections from Puppeteer or other clients, and it does **not** restore CDP in Firefox. It gives *your Python code* a small, CDP-*style* `Tab` API that's backed by Firefox's **WebDriver BiDi**. If you've used a raw CDP `Tab`, this feels identical to use — but "CDP-style" describes the shape of the API, not the wire protocol underneath (which is BiDi).

II. WHAT ACTUALLY HAPPENED TO CDP IN FIREFOX

Firefox shipped an experimental CDP implementation back in 2021 (Firefox 86), so tools like Puppeteer could talk to it. Then Mozilla decided to stop maintaining a protocol it didn't control and standardize on the W3C's WebDriver BiDi instead:

- Firefox 129 (May 2024): CDP support deprecated — off by default, still opt-in via the remote.active-protocols pref.
- Firefox 141 (June 2025): CDP completely removed , pref and all. 141 is the first Firefox where you simply cannot use CDP.

So on any current Firefox, point `--remote-debugging-port` at it expecting CDP and the `/json/*` discovery endpoints return **HTTP 404**. The Remote Agent is still there — it just speaks **WebDriver BiDi** now. (Verified live on Firefox Developer Edition 153.)

This transition is worth describing precisely: WebDriver BiDi is not "CDP with new names." It is a different standardized bidirectional protocol with a different command and event model and feature surface. This adapter exposes a CDP-like Python interface over the BiDi wire protocol; it does not restore native CDP or claim complete CDP compatibility.

III. WHY A 700-LINE ADAPTER INSTEAD OF "JUST USE PLAYWRIGHT"

Fair question — and the answer is scope, not competition. Playwright, Selenium and Puppeteer are frameworks: driver

binaries, big dependency trees, an execution model. If you're writing end-to-end tests, use them.

But if you're writing a small automation *script* or agent that already thinks in terms of a minimal `Tab` object, pulling in a whole framework just to get a Firefox backend is a bad trade. This library is for that case: raw websocket, one pip dependency, uses the Firefox already on the machine, and reads top to bottom as a clean reference for how BiDi actually works. A library, not an ecosystem.

IV. THE FOUR THINGS THAT WILL 403 OR 404 YOU

Getting a BiDi session up has a few non-obvious traps. The adapter handles all of them; here they are so the protocol isn't a black box:

1. The endpoint has a path. The agent logs WebDriver BiDi listening on ws://127.0.0.1:<port> — with no path . The bare root is just an HTTP server. The real websocket is at ws://<port>/session .
2. session.new is mandatory. Send session.new {capabilities:{}} before any browsingContext.* or script.* call. Its reply carries the real browserVersion .
3. There's an Origin allowlist. websocket-client sends Origin: http://127.0.0.1:<port> , so launch with -remote-allow-origins http://127.0.0.1:<port> or the handshake dies with 403 "incorrect Origin header" before you send a command.
4. script.evaluate returns a BiDi remote value , not plain JSON — {"type":"string","value":"…"} , arrays and objects as tagged shapes. A small deserialize_bidi() walks it back into native Python.

V. WHAT USING IT LOOKS LIKE

```
from firefox_bidi_worker import launch_firefox,
FirefoxBiDi

proc, ws_url, profile = launch_firefox(port=9227)   #
isolated, visible Firefox
ff = FirefoxBiDi(ws_url)                             #
auto-creates the BiDi session

ff.navigate("https://example.com")
print(ff.browser_version)                            #
"153.0"
print(ff.evaluate("document.title"))                 #
"Example Domain"
ff.screenshot("shot.png")
ff.type_text("hello"); ff.click_xy(640, 400)         #
real, trusted input
```

Every launch uses `-no-remote` plus a dedicated profile, so it's a fully isolated instance — it never attaches to, mutates, or kills the Firefox you already have open. There's a unit test that asserts those isolation flags can't be quietly removed, and the whole pipeline is self-proving: `python firefox_bidi_worker.py --selftest` runs launch → session → navigate → read title → screenshot and reports green, with a 19-test suite locking the protocol contracts.

(The repo also ships a small livestream-tooling bonus — SEO/overlay/clip helpers and an obs-websocket client I built alongside it — but the core is the BiDi worker above.)

If you maintain a browser worker and want a Firefox backend without adopting a framework, or you just want a readable reference for how WebDriver BiDi works over a raw socket, it's here, MIT-licensed: **github.com/Zek21/firefox-cdp**. Issues welcome, especially if the launch handshake differs on your Firefox version.

VI. LIMITATIONS AND CLAIM BOUNDARY

The adapter uses a single-session API and does not serve incoming CDP connections or act as a network proxy. The source evidence is implementation/self-test based rather than a comprehensive compatibility benchmark.

Claim boundary - State clearly that the library provides a CDP-like Python interface over the BiDi wire protocol; it does not restore native CDP or claim complete CDP compatibility.

VII. REFERENCES

Source links and citations are preserved in the evidence record above.

AUTHOR BIOGRAPHY

EXZIL CALANZA is an independent researcher and software builder based in Iloilo City, Philippines. His work focuses on browser and desktop automation, reproducible evaluation, agent reliability, workflow verification, and AI-assisted systems engineering.