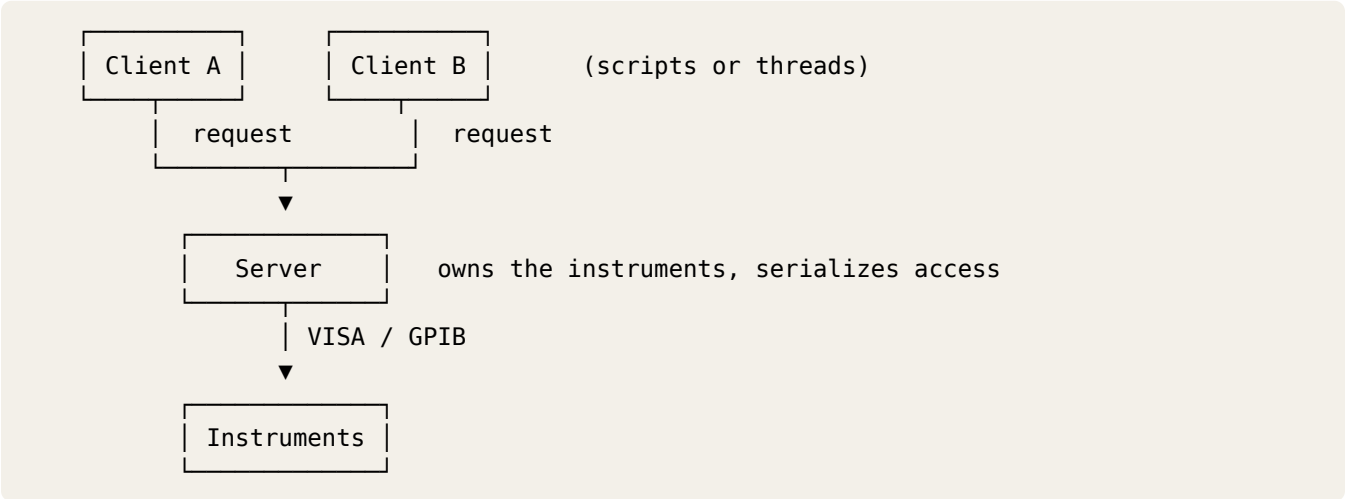
Sharing an Instrument Across Processes

A single VISA or GPIB bus can only handle one conversation at a time. The moment two parts of your setup want to talk to instruments simultaneously — say, one loop logging temperature while another sweeps a voltage, or a GUI reading a meter while a background thread records data — their commands collide on the bus and you get timeouts, garbled replies, or worse.

The fix is to put a **server** in front of the hardware. One process owns the instruments and is the only thing that touches the bus; everything else becomes a **client** that sends requests to the server. The server handles requests one at a time, so clients automatically queue instead of colliding.

> Prerequisites
>
> Instrument Wrapper Classes (the server holds `Device` objects) and basic familiarity with functions and classes. This is an advanced pattern — reach for it only when you genuinely need concurrent access.

1 The Idea



Client and server talk over a **socket** — a two-way network connection. We run both on the same computer (`localhost`), so no real network is involved; the socket is just a clean way for separate processes to exchange messages.

2 A Warm-Up: Echo Server

Before wiring in hardware, confirm the socket machinery works with a server that just echoes messages back reversed. This is the whole socket pattern in miniature.

```python
# server.py
import socket


def run_echo_server(host="localhost", port=62538):
    running = True
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
        s.bind((host, port))
        s.listen()
        print(f"Echo server listening on port {port}")
        while running:
            conn, addr = s.accept()
            with conn:
                while True:
                    data = conn.recv(1024)
                    if not data:                # client hung up
                        break
                    if data == b"shutdown":
                        running = False
                        break
                    conn.sendall(data.decode()[::-1].encode())  # reversed


if __name__ == "__main__":
    run_echo_server()
```

```
# client.py
import socket


def send(msg, host="localhost", port=62538):
    if not msg:
        return "did not send anything"
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        s.send(msg.encode())
        return s.recv(1024).decode()


if __name__ == "__main__":
    print(send("ping"))   # -> "gnip"
```

Run `python server.py` in one terminal, then `python client.py` in another. The two key socket moves are **`bind` + `listen` + `accept`** on the server and **`connect` + `send` + `recv`** on the client. Everything below just swaps the “reverse the string” step for “talk to an instrument.”

3 The Instrument Server

Now make the server own real instruments and route requests to them. A simple text protocol keeps client messages readable:

```
<device-id>::<command>::<optional payload>
```

where `<command>` is write (`W`), read (`R`), or query (`Q`).

```
# server.py
import socket
from instruments import Device   # your wrapper class (see Instrument Wrapper Classes)
```

```
class InstrumentServer:
    shutdown_command = b"shutdown"

    def __init__(self, host="localhost", port=62538):
        self.address = (host, port)
        self.running = False
        # One Device per instrument on the bus. Keys are the IDs clients will use.
        self.devices = {
            "TC": Device("GPIB0::12::INSTR"),   # temperature controller
            "VS": Device("GPIB0::4::INSTR"),    # voltage supply
        }

    def handle(self, message: str) -> str:
        """Parse '<id>::<command>::<payload>' and act on the named device."""
        parts = message.upper().split("::")
        dev_id, command = parts[0], parts[1]
        payload = parts[2] if len(parts) > 2 else ""

        device = self.devices.get(dev_id)
        if device is None:
            return f"unknown device id: {dev_id}"

        if command.startswith("W"):
            device.write(payload)
            return "sent"
        elif command.startswith("Q"):
            return device.query(payload)
        elif command.startswith("R"):
            return device.read()
        return f"unknown command: {command}"

    def run(self):
        self.running = True
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.bind(self.address)
            s.listen()
            print(f"Instrument server listening on port {self.address[1]}")
            while self.running:
                conn, addr = s.accept()
                with conn:
                    while True:
                        data = conn.recv(1024)
                        if not data:
                            break
                        if data == InstrumentServer.shutdown_command:
                            self.running = False
                            break
                        conn.sendall(self.handle(data.decode()).encode())


if __name__ == "__main__":
    InstrumentServer().run()
```

Because `handle()` runs inside the server's single accept loop, requests are processed strictly one at a time — that serialization is exactly what prevents bus collisions.

4 A Client That Looks Like a `Device`

You *could* make every caller build `"TC::Q::KRDG? A"` strings by hand, but it's far nicer to give clients an object with the same method names as the `Device` wrapper. Then code can switch between talking directly to hardware and talking through the server with a one-line import change.

```
# client.py
import socket


def send(msg, host="localhost", port=62538):
    if not msg:
        return "did not send anything"
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        s.send(msg.encode())
        return s.recv(1024).decode()


class DeviceClient:
    """Mirrors the Device interface, but sends requests through the server."""

    def __init__(self, dev_id, host="localhost", port=62538):
        self.dev_id = dev_id
        self.host = host
        self.port = port

    def write(self, msg):
        return send(f"{self.dev_id}::W::{msg}", self.host, self.port)

    def read(self):
        return send(f"{self.dev_id}::R", self.host, self.port)

    def query(self, msg):
        return send(f"{self.dev_id}::Q::{msg}", self.host, self.port)

    def get_id(self):
        return self.query("*IDN?")
```

A `DeviceClient("TC")` is now used just like a `Device`:

```
tc = DeviceClient("TC")
print(tc.query("KRDG? A"))   # routed through the server to the real instrument
```

You can even subclass `DeviceClient` the same way you subclassed `Device` — a `TemperatureController` client with a `temperature()` method — so the physics-level interface is identical whether or not a server sits in the middle.

5 Which Approach to Use

	Direct (`Device`)	Through a server (`DeviceClient`)
Setup	Just run your script	Start the server first, in its own process
Concurrent access	One script/thread only	Many scripts/threads safely
Complexity	Low	Higher — two processes to manage

Use direct access for ordinary work where one script owns the instruments. **Add the server** only when you genuinely need concurrency — for example, an acquisition loop and a control GUI hitting the same bus at once.

6 Where This Leads

- **Structuring a Data-Acquisition App** runs an acquisition loop and a server together using threads.
- **Building a GUI** is a common reason to need concurrent access — the interface stays responsive while data collection runs in the background.

Back to Python for Lab Work