Power Supply Driver Design

Abe Spitalny

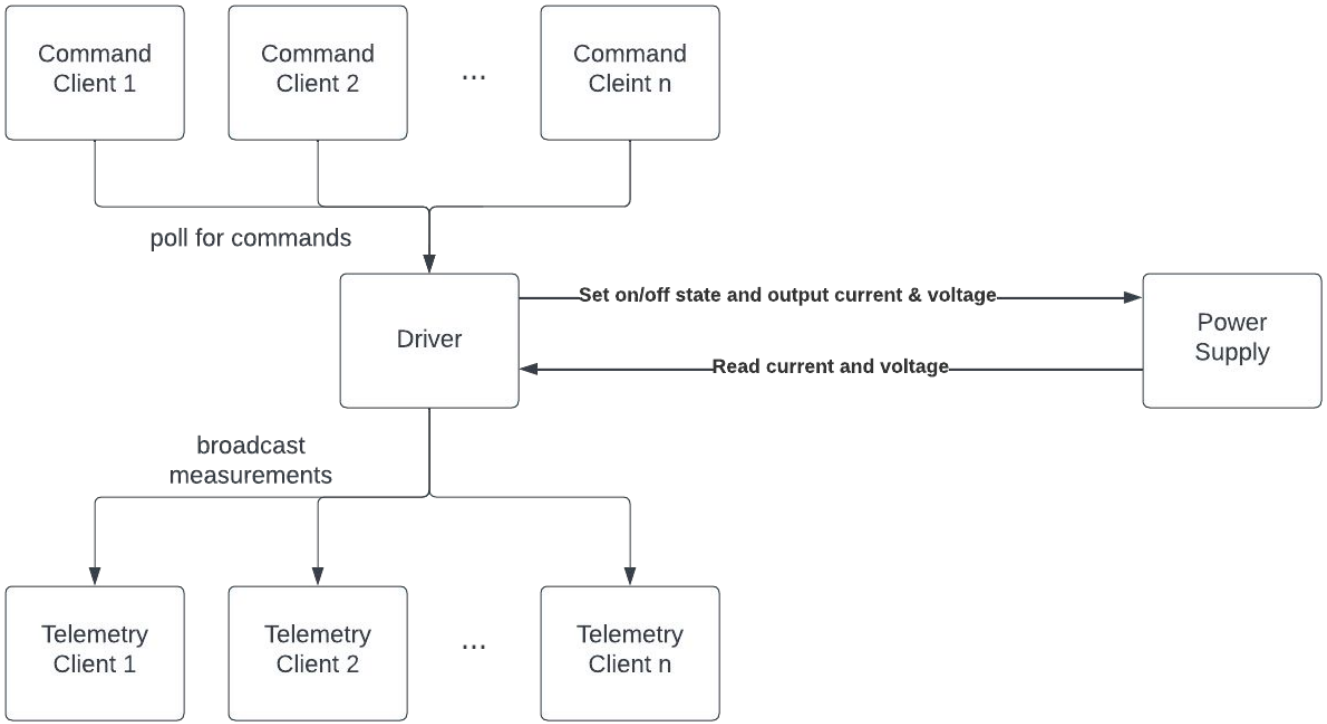
Roadmap

- The problem
- High-level architecture
- The moving parts
- Design decisions and trade-offs
- Features for the future
- Review solution

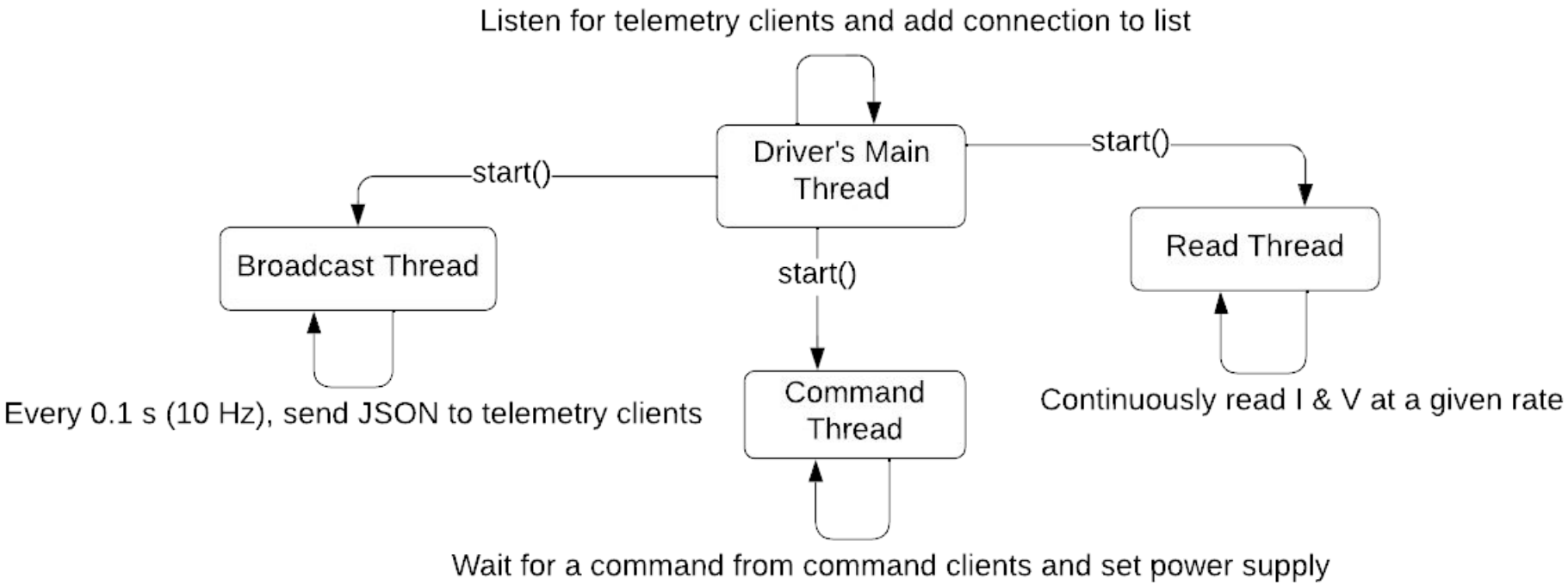
The Problem

- Create power supply driver to read measurements and set output. Generally, a driver is located in kernel to provide an interface for OS to read and write to external devices.
- Use SCPI to read and write to power supply over USB <-> Serial RS-232
- Can specify different rates which driver can read measurements.
- Listen for telemetry clients and send power supply measurements at a specified interval over a TCP socket.
- Specified command clients can set power supply output over a TCP socket.
- Reading and writing to power supply shouldn’t block telemetry broadcast.

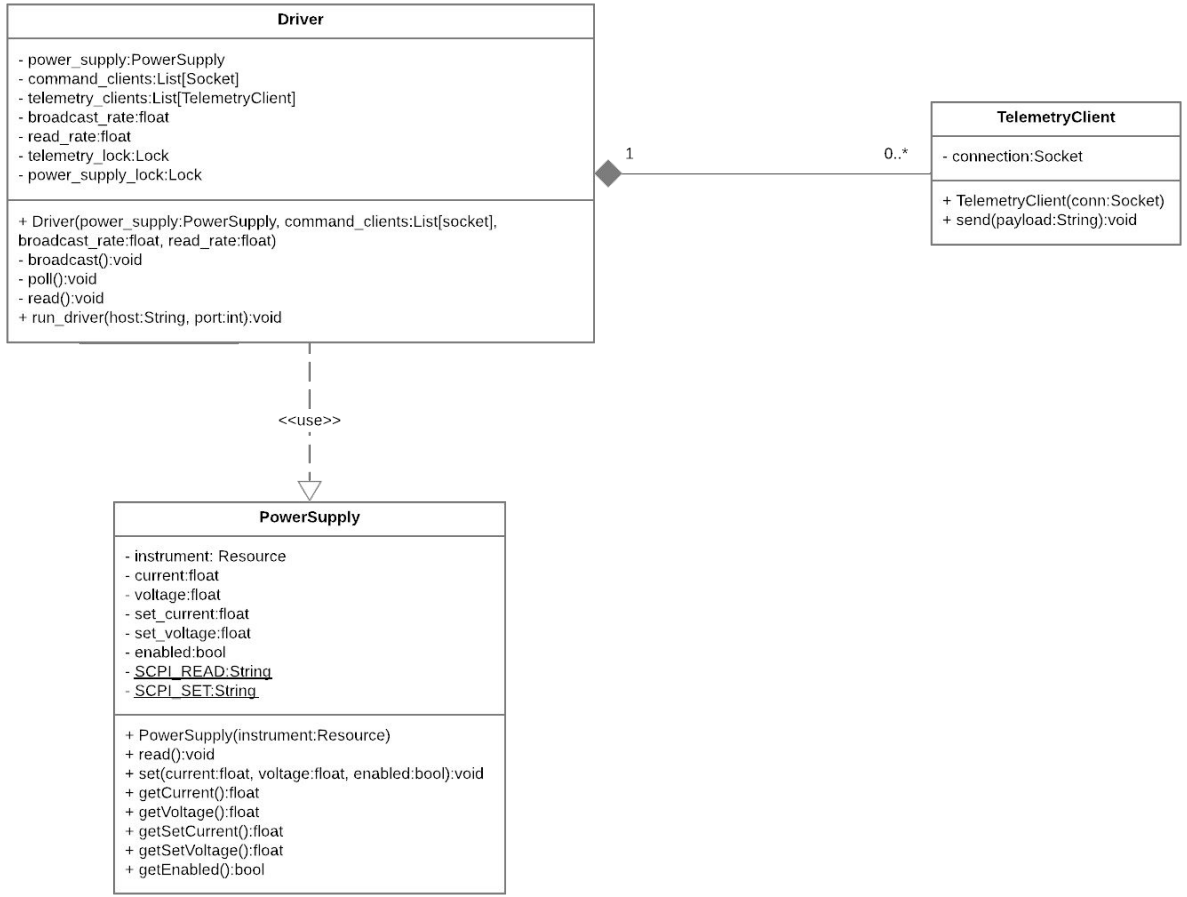
High-level architecture



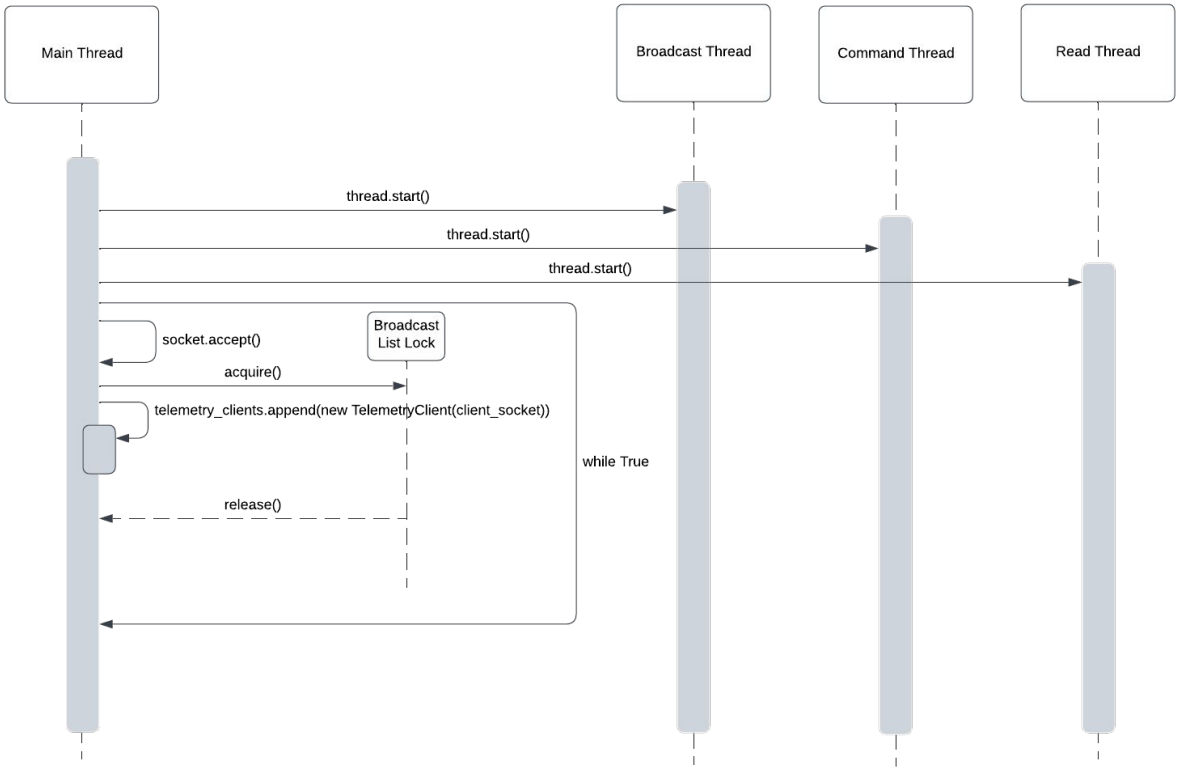
High-level architecture



The moving parts: UML Diagram



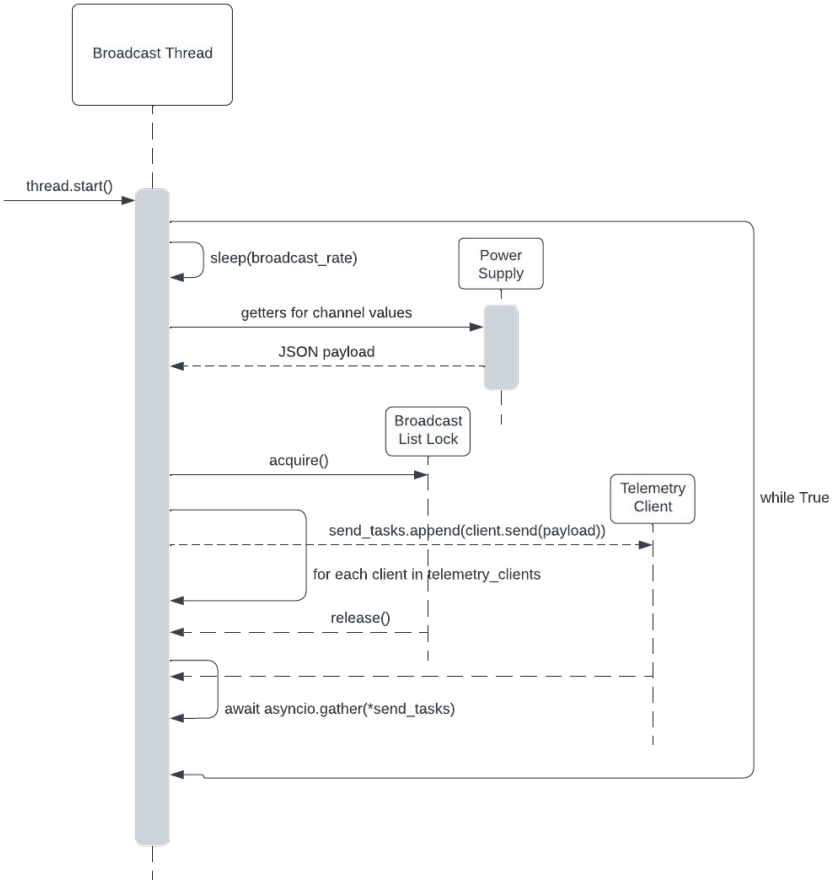
The moving parts: Sequence Diagram of Main Thread



The moving parts: Pseudocode of Main Thread

```
def run_driver(host, port):
        # Start up threads
        broadcast_thread = threading.Thread(target=broadcast)
        broadcast_thread.start()
          …
        # Listen for connections from telemetry clients.
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.bind((host, port))
        sock.listen()
        while True:
                client_sock, _ = sock.accept()
                # Use lock when adding new telemetry client to broadcast list
                # because the list is shared by the broadcast thread.
                with telemetry_lock:
                        telemetry_clients.append(TelemetryClient(client_sock))
```

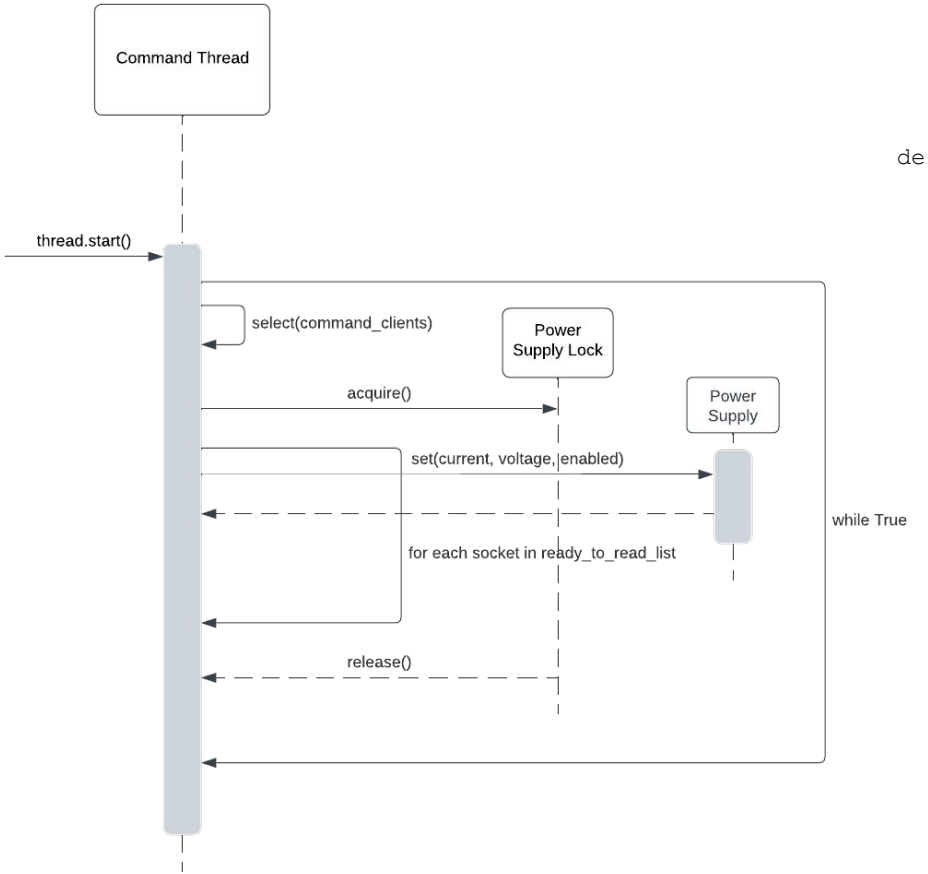
The moving parts: Sequence Diagram of Broadcast Thread



The moving parts: Pseudocode of Broadcast Thread

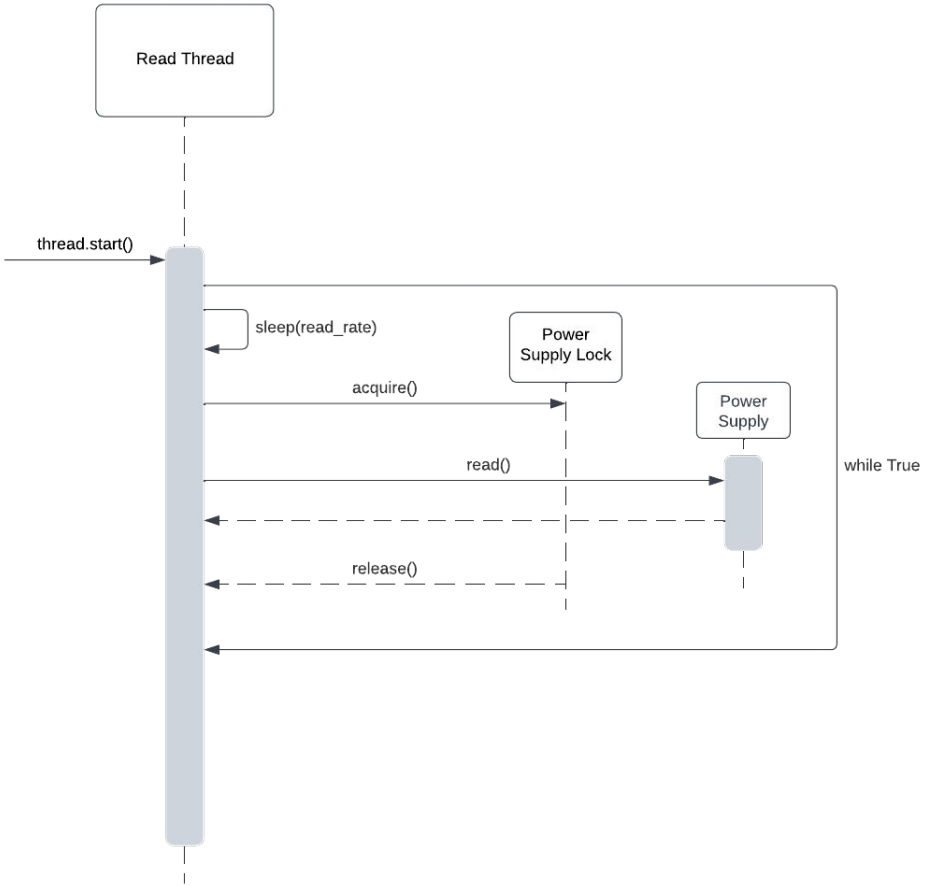
```
async def broadcast():
      while True:
            time.sleep(broadcast_rate)  # In our case, the broadcast rate is 0.1 seconds or 10 Hz.
            payload = {
                  "current": power_supply.current,
                  "voltage": power_supply.voltage,
                  "set_current": power_supply.set_current,
                  "set_voltage": power_supply.set_voltage,
                  "enabled": power_supply.enabled
            }
            payload = json.dumps(payload).encode("utf-8")  # Convert dictionary object to JSON string.
            with telemetry_lock:  # Get lock because we're iterating over a shared list.
               send_tasks = [client.send(payload) for client in telemetry_clients]

            await asyncio.gather(*send_tasks)  # Wait for all telemetry clients to receive measurements.
```

Sequence Diagram and Pseudocode of Command Thread




```
def poll():

        while True:

                rlist, _, _ = select.select(command_clients, [], [])

                with power_supply_lock:

                        for sock in rlist:

                                payload =
                        json.loads(sock.recv(4096).decode("utf-8"))

                                power_supply.set(payload["set_current"],
                        payload["set_voltage"], payload["enabled"])
```

Sequence Diagram and Pseudocode of Read Thread




```
def read():
      while True:
            time.sleep(read_rate)

            with power_supply_lock:
                  power_supply.read()
```

The moving parts: Brief mention of SCPI commands

- Defined as static constants in PowerSupply class
- Query command = "MEAS:CURR:DC?;MEAS:VOLT:DC?"
- Set command = "CURR {};VOLT {};OUTP:{}"
- SCPI commands are ASCII strings
- Can concatenate multiple commands using ‘;’
- SCPI commands can take arguments, e.g., CURRent can take a number which is used to set the current output of the device.
- Arguments don’t have to be numbers, e.g., OUTPut takes a string argument of either ‘START’ or ‘STOP’.
- Query commands are indicated by a ‘?’ at the end.

Design decisions and trade-offs

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

- multiprocessing vs. multithreading vs. asyncio
  - Went with multithreading despite GIL because parallelism depends on hardware, and threads are lightweight.
  - Used asyncio for parallelizing broadcast because it's mainly I/O.
- Lock shared PowerSupply variables in broadcast thread
  - Did not use lock because broadcast could block for reads/writes.
  - Could use reader-writer pattern where we prioritize readers, but could starve writers.
  - Opens us up to rare race conditions.
- Use atomic variables in PowerSupply
  - Updating a variable is an atomic operation.
- Have thread continuously read from power supply
  - Could read from power supply only after writing to it, but this assumes it can't change via other means.
  - Input to driver to specify read rate.

Features for the future

- Handle clean up and errors.
- Telemetry client can subscribe to get only certain channel values.
- Additional SCPI commands.
- Prevent race conditions by locking power supply in broadcast thread and implement fair scheduling using a queue + aging approach.

Review solution

- ✓ Read and write functions must be non-blocking for telemetry broadcast
  - Reading from and writing to power supply both happen in their own thread separate from the broadcast thread, and they don’t wait on each other’s shared resources.
- ✓ Read rate must be capable of being much higher or lower than the broadcast rate
  - Read rate is independent from broadcast rate. This wouldn’t be the case if we locked the power supply in the broadcast thread which would lead to either thread being starved depending on the read rate.
- ✓ Broadcast telemetry to multiple telemetry client connections at 10 Hz
- ✓ Receive commands from remote command clients

Questions?