Sensor Stream Aggregator

Design and Analysis Report

Nipun Pal

Embedded Systems Engineer

November 2025

Contents

1 Problem Understanding

The assignment aimed to implement a complete client-side system capable of:

- Reading sensor data streams from three TCP ports.
- Understanding their signal properties.
- Implementing a control loop through a binary UDP protocol.
- Documenting observations and reasoning.

The challenge intentionally lacked protocol specifics, requiring systematic investigation and assumptions guided by experimental validation.

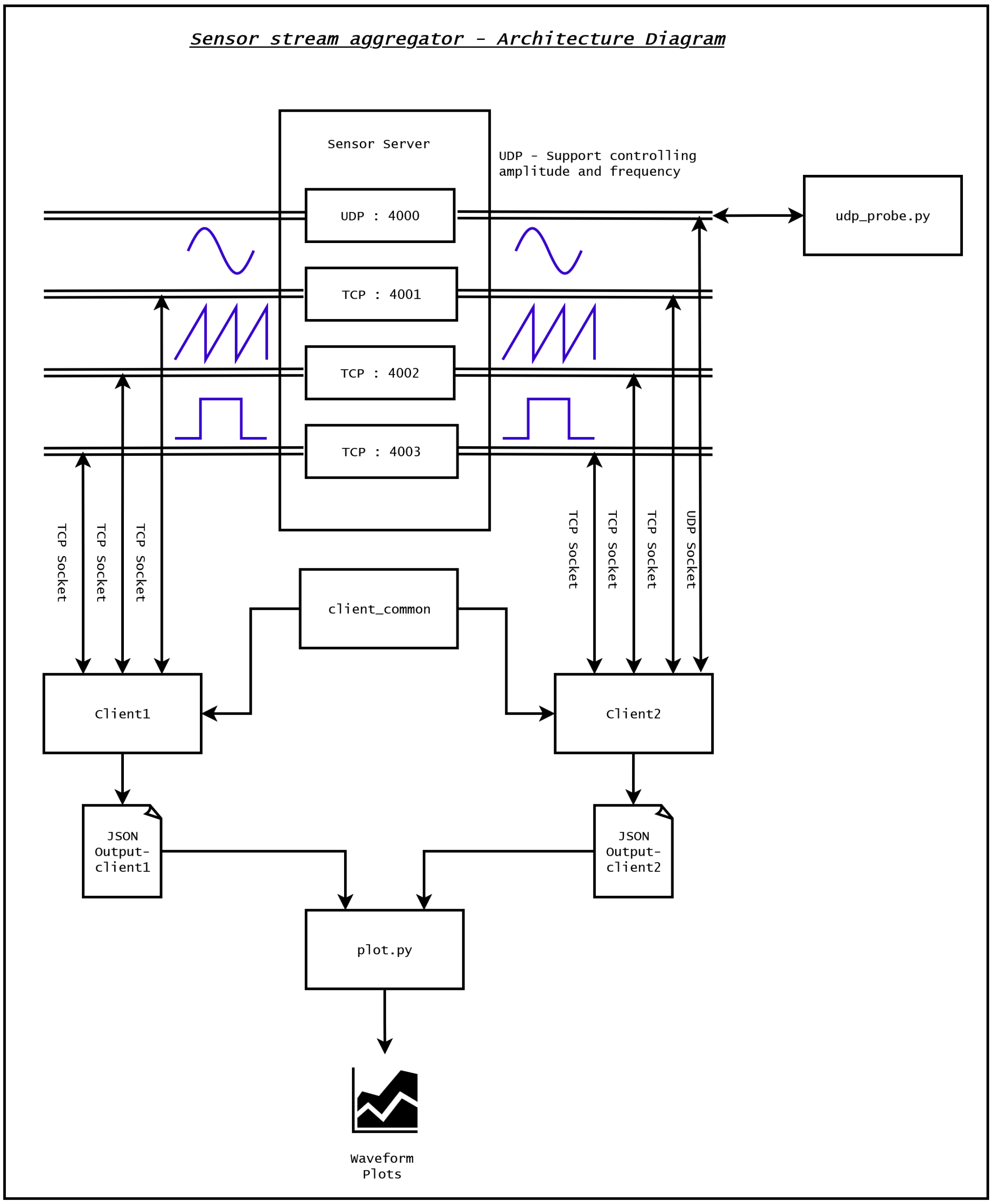
2 Approach and Incremental Development

The solution was built iteratively in small, verifiable steps:

1. **Stage 1 – Basic probe:** Created `probe.c` to connect to TCP ports (4001–4003) and print raw values. This confirmed that data streams were ASCII-formatted numbers representing real-time sensor readings.
2. **Stage 2 – Aggregator (client1):** Built `client1.c` to perform non-blocking reads from all ports and log the latest value of each every 100 ms in structured JSON. This enabled correlation between signals and served as the baseline data collection client.
3. **Stage 3 – Visualization (plot.py):** Developed `tools/plot.py` to parse JSON and visualize the waveforms.
4. **Stage 4 – UDP protocol discovery (udp_probe.py):** Implemented a UDP probing tool to test different object-property combinations and note server responses. This revealed the mapping between property IDs and their functions (frequency, amplitude, enable flags, durations, etc.).
5. **Stage 5 – Scaling analysis:** By comparing waveform amplitudes and frequencies from `client1` with values written via UDP, scaling factors were determined:
   - Frequency: 500 units $\approx$ 1 Hz
   - Amplitude: values in mV (5000 $\to$ 5 V)
   - Pluse Width Duration: 1000–10000 $\approx$ 1–10 s
6. **Stage 6 – Closed-loop controller (client2):** Implemented `client2.c` to read all outputs every 20 ms, and whenever `out3 >= 3.0`, adjust `out1` parameters via UDP. When `out3 < 3.0`, restore frequency and amplitude accordingly.

7. **Stage 7 – Validation:** Logged JSON output from `client2` and plotted it using the same visualization pipeline. Observed transitions matched the expected control behavior.

3 System Architecture



4 System Overview

The entire setup operates within the provided Docker environment. The Docker container (`fsw-linux-homework`) runs a server application that exposes three TCP ports-4001, 4002, and 4003-and a UDP port (4000). The TCP ports continuously stream numeric waveform data representing three distinct outputs (`out1`, `out2`, and `out3`), while the UDP port serves as a control interface for reading and modifying signal properties.

4.1 Overall Architecture

- The **Sensor Server** acts as the data source and signal generator.
- The **Client Suite**-comprising `probe.c`, `client1.c`, `client2.c`, and shared utilities in `client_common.c`-implements data capture, analysis, and control functionalities.
- Python tools (`plot.py` and `udp_probe.py`) are used for waveform visualization and property discovery.

All development and verification were performed against the running Docker instance using port mappings:

```
docker run -p 4000:4000/udp -p 4001:4001
      -p 4002:4002 -p 4003:4003 fsw-linux-homework
```

4.2 Client1 - Data Acquisition and Logging

`client1.c` implements the baseline monitoring functionality. Its objective is to read continuously from all three TCP data streams and generate structured JSON logs every 100 ms.

- Each 100 ms cycle captures the most recent value received from each TCP port (4001–4003).
- If multiple values arrive within the same time window, only the **latest** value is retained.
- If no data is received within the window, the corresponding output field is set to the string `"--"`.
- The generated line is a single JSON object containing four fields: `timestamp`, `out1`, `out2`, and `out3`.
- No aggregation or buffering occurs beyond the current window; each JSON object is printed independently on standard output.

Example output:

```
{"timestamp": 1709286246830, "out1": "-4.8", "out2": "8.0", "out3": "--"}
{"timestamp": 1709286246930, "out1": "-4.0", "out2": "--",  "out3": "1.0"}
{"timestamp": 1709286247030, "out1": "-2.9", "out2": "1.2", "out3": "--"}
```

This client serves as the foundation for understanding signal properties such as frequency, amplitude, and shape. The resulting JSON logs are visualized via `plot.py`, revealing three distinct waveforms: sine (out1), triangle (out2), and square (out3).

4.3 Client2 - Closed-loop Control and Monitoring

`client2.c` builds upon `client1`'s data acquisition framework and adds a feedback control mechanism via the UDP channel (port 4000).

- The main loop executes every 20 ms, reading the most recent values from the same TCP sources.
- The JSON logging behavior remains identical to `client1`, but with 5× higher resolution.
- In parallel, the client monitors the value of `out3` and adjusts the parameters of `out1` through the binary UDP protocol as follows:
  - When `out3 >= 3.0`: set `out1 → freq = 1 Hz, amp = 8000 mV`.
  - When `out3 < 3.0`: set `out1 → freq = 2 Hz, amp = 4000 mV`.
- UDP messages follow the server's 16-bit big-endian protocol structure with four fields: `operation`, `object`, `property`, and `value`.
- The control logic executes asynchronously, ensuring timing precision and uninterrupted data logging.

In operation, `client2` dynamically regulates the waveform properties of `out1` based on the digital threshold behavior of `out3`. The logged output and plots confirm the correctness of the behavior, with out1 transitions between high and low frequency states aligning with `out3` thresholds set depending on out3 value.

5 Design Decisions

- **Timing:** precise 100 ms (client1) and 20 ms (client2) loops implemented via `now_ms()` and select-based waiting.
- **Non-blocking I/O:** all TCP sockets configured as non-blocking to allow concurrent reads without threading.
- **Error Recovery:** automatic reconnection of TCP streams and UDP resend logic on communication loss.

- **Data Representation:** JSON chosen for portability and ease of plotting and analysis.
- **Visualization:** `plot.py` Fills for missing data points for smooth temporal continuity.

6 Protocol and Property Mapping

6.1 Binary Control Protocol

- 16-bit unsigned integer fields.
- Big-endian byte order.
- Operations:
  - 1 = read (3 fields)
  - 2 = write (4 fields)A

6.2 Property IDs and Meanings (discovered empirically)

Output Channel	Property ID	Meaning
4001 / 4002	14	Enable flag
4001 / 4002	170	Amplitude (mV)
4001 / 4002	255	Frequency (Hz units)
4003	14	Enable flag
4003	42	Minimum duration
4003	43	Maximum duration

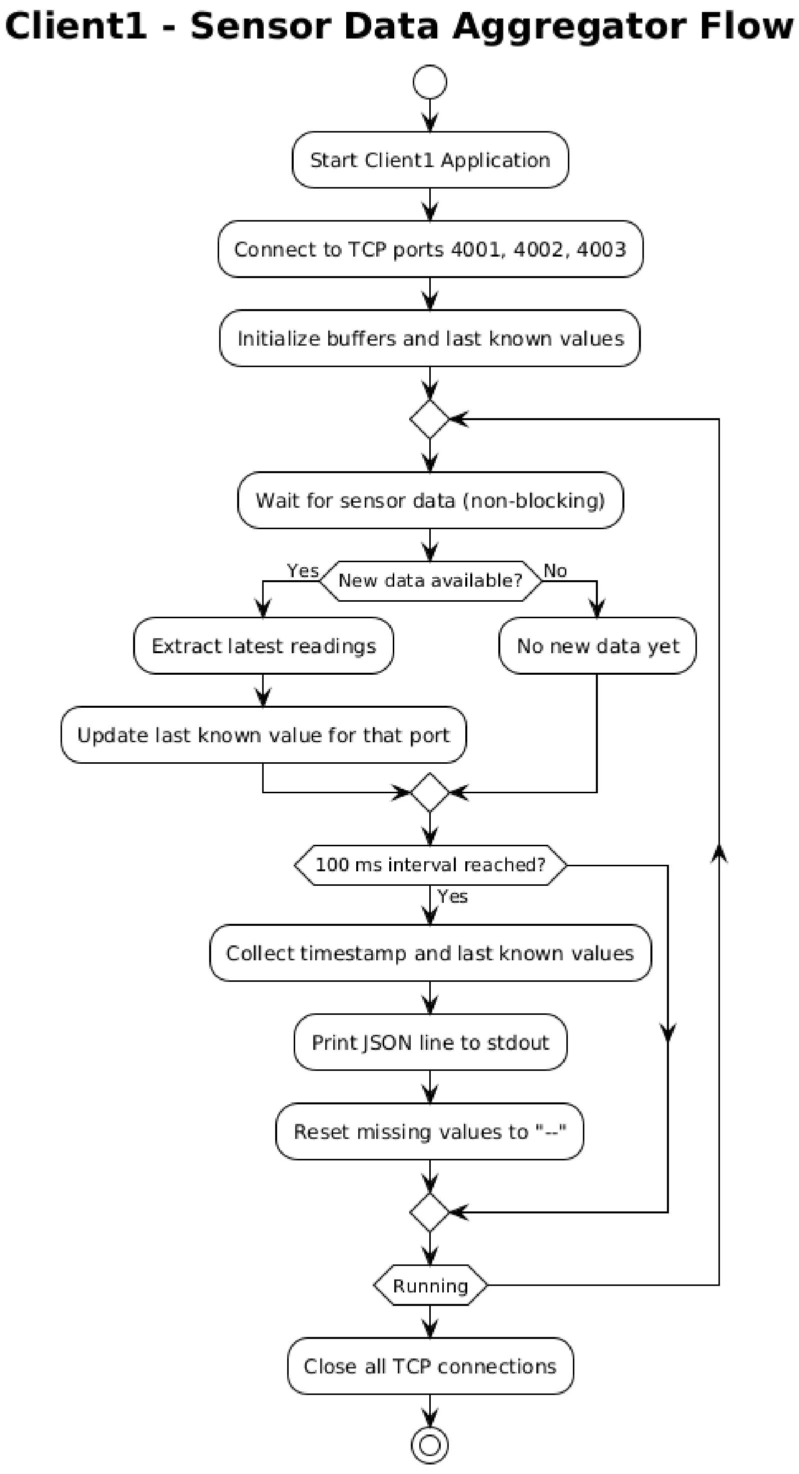
Table 1: Property ID Mapping and Functional Meaning

6.3 Value Ranges (determined via UDP probe)

Parameter	Range	Interpreted Value Range
Amplitude	1000–10000	(+/-)1 V to (+/-)10 V (expressed in mV)
Frequency	50–2000	0.1–4 Hz (500 approx 1 Hz)
Duration	1000–10000	1–10 s

7 Flowcharts

Below are in brief flowcharts for client1 and client2 respectively.



Client1 - Sensor Data Aggregator Flow
Start Client1 Application
Connect to TCP ports 4001, 4002, 4003
Initialize buffers and last known values
Wait for sensor data (non-blocking)
New data available?
Yes
No
Extract latest readings
No new data yet
Update last known value for that port
100 ms interval reached?
Yes
Collect timestamp and last known values
Print JSON line to stdout
Reset missing values to "--"
Running
Close all TCP connections

Figure 1: Flowchart of Client1 - Sensor Data Aggregator

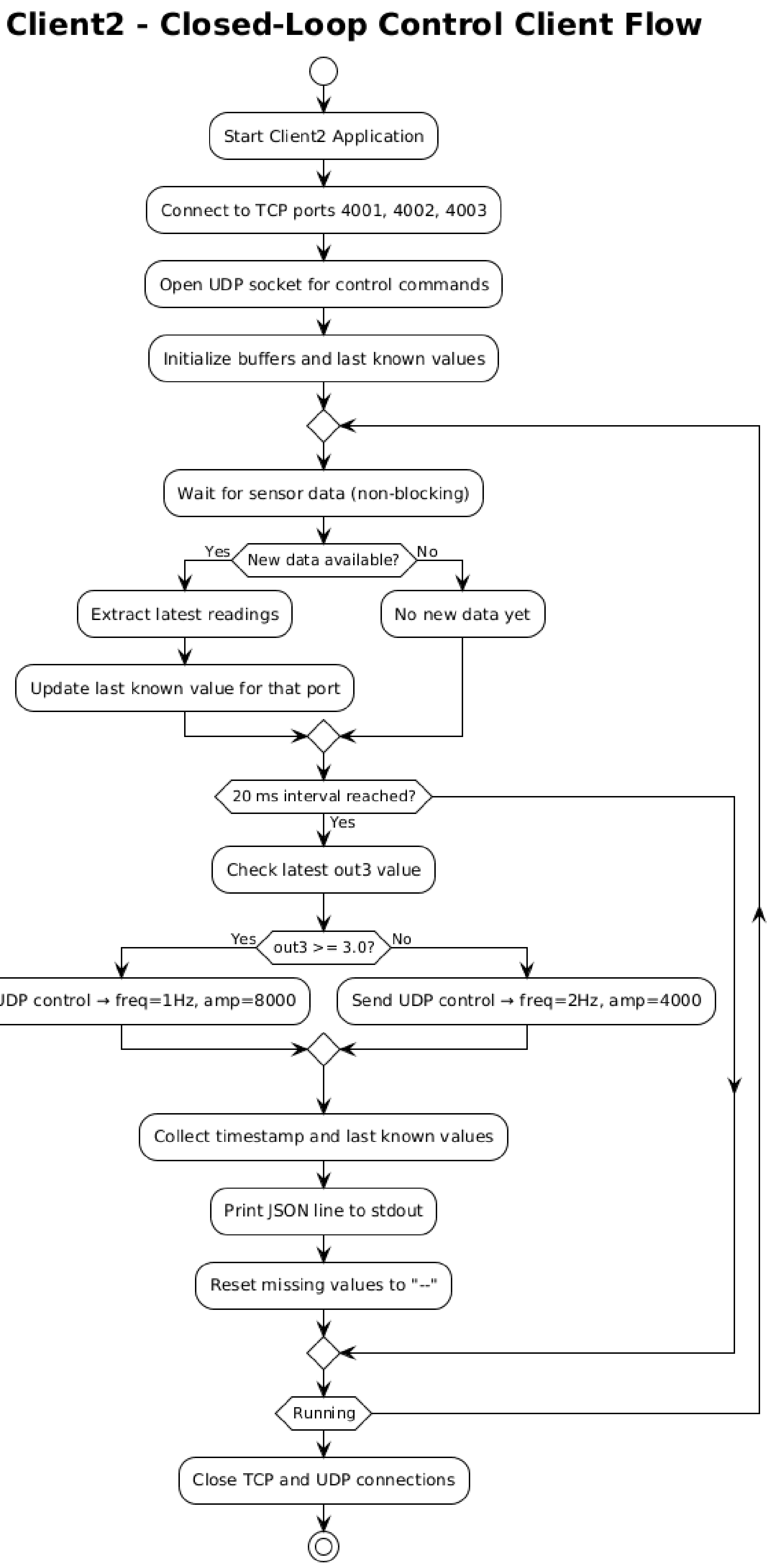

Client2 - Closed-Loop Control Client Flow
Start Client2 Application
Connect to TCP ports 4001, 4002, 4003
Open UDP socket for control commands
Initialize buffers and last known values
Wait for sensor data (non-blocking)
New data available?
Yes
No
Extract latest readings
No new data yet
Update last known value for that port
20 ms interval reached?
Yes
Check latest out3 value
out3 >= 3.0?
Yes
No
Send UDP control → freq=1Hz, amp=8000
Send UDP control → freq=2Hz, amp=4000
Collect timestamp and last known values
Print JSON line to stdout
Reset missing values to "--"
Running
Close TCP and UDP connections

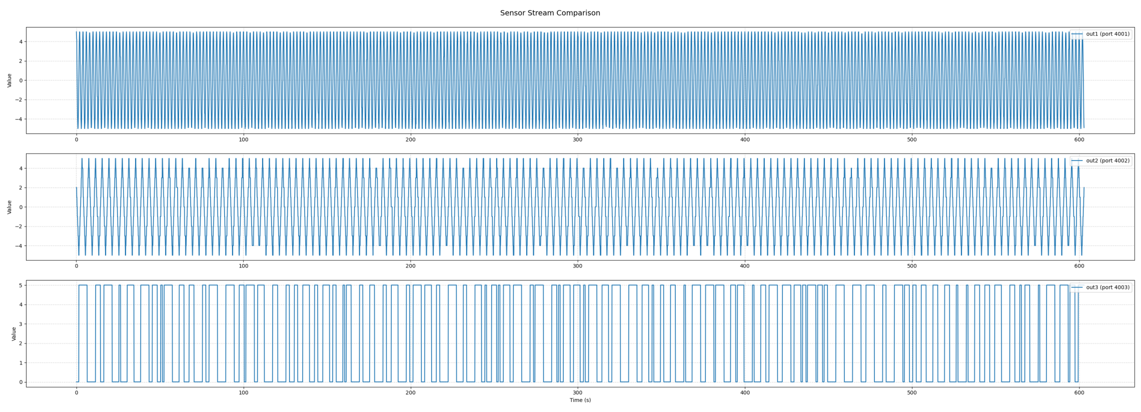
Figure 2: Flowchart of Client2 - Closed-Loop Control Client

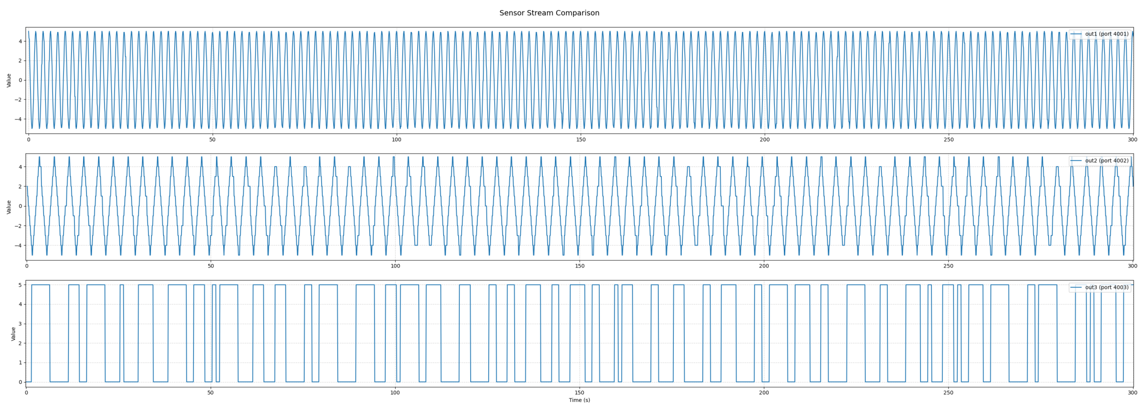
8 Observations

8.1 Client1 – Monitoring Only (100 ms Sampling) (Data capture duration = 10mins)

Output	Waveform Type	Observed properties
out1 (4001)	Sine wave	Smooth waveform, amplitude (+/-)5 V, 1 Hz
out2 (4002)	Triangle wave	Slower waveform, 0.5 Hz
out3 (4003)	Square wave	Digital output (0–5 V), 8–10 s period

Table 2: Observed properties of Output Streams (Client1)



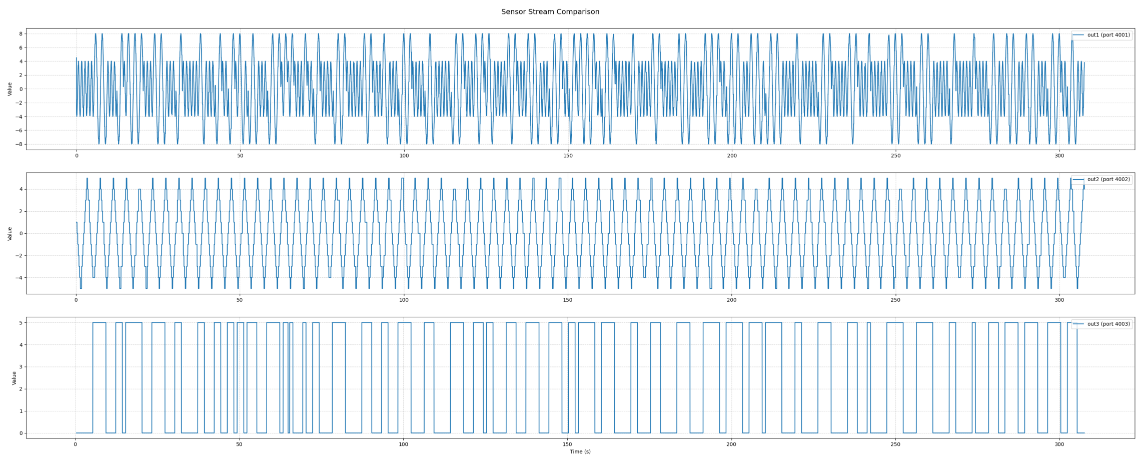


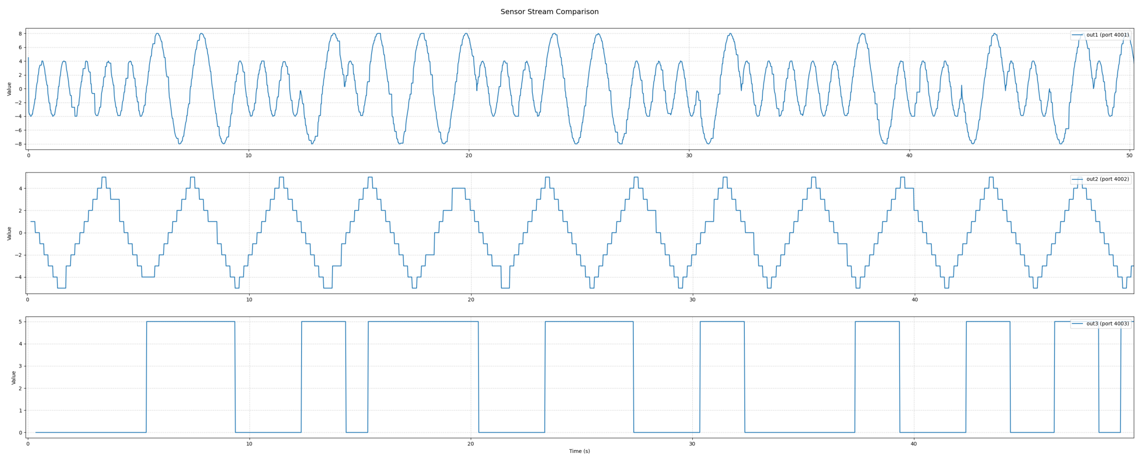
8.2 Client2 – Closed-loop Control (20 ms Sampling) (Data capture duration = 5mins)

Logic implemented:

- When `out3 >= 3.0`: set `out1 → freq = 1 Hz, amp = 8000 mV`.
- When `out3 < 3.0`: set `out1 → freq = 2 Hz, amp = 4000 mV`.

Observed transitions in waveform confirm these actions.





9 Correctness and Verification

1. UDP probe confirmed correct property IDs and ranges.
2. Plot of `out1` clearly changes frequency and amplitude when `out3` toggles, verifying logic.
3. Time drift tests over several minutes show steady 20 ms periodicity.
4. Control transitions observed in zoomed graph perfectly align with `out3` thresholds.

Fault Tolerance Summary

Failure Type	**Detection**	**Recovery**	**Outcome**
TCP disconnect	`read_extract_latest()` returns <0	Reconnect via `connect_nonblocking()`	Self-recovers without restart
UDP send error	`sendto()` short write	Retry after delay	Recovers if transient
Missing data	No message within sampling window	Substitute "–" placeholder	Prevents malformed JSON output
Program exit	Manual or error termination	Closes all sockets and descriptors	No file descriptor leaks
Delayed samples	Controlled by 20 ms / 100 ms window	Non-accumulative timing logic	Output remains periodic

10 Conclusion

The development followed a measured investigative approach:

1. Observe, measure, and visualize raw signals.
2. Reverse-engineer protocol semantics through experimentation.
3. Validate assumptions through closed-loop response.

client1 provides the data acquisition baseline, while **client2** adds the control logic and achieves full compliance with exercise requirements. Graphs demonstrate stable signal acquisition, accurate scaling, and correct feedback loop operation.