Chapter 4
Coordinating Cross-Process Sensor Aggregation in Rust With IPC and Shared Memory

Haochen Shi
http://orcid.org/0009-0004-1165-1682
The Hong Kong Polytechnic University, Hong Kong

Xuan Luo
The Hong Kong Polytechnic University, Hong Kong

Junhao Huang
The Hong Kong Polytechnic University, Hong Kong

Yixiong Feng
The Hong Kong Polytechnic University, China

Zihan Meng
The Hong Kong Polytechnic University, Hong Kong

Aquil Mirza Mohammed
http://orcid.org/0000-0001-7756-4363
The Hong Kong Polytechnic University, Hong Kong

ABSTRACT

This chapter, per the authors, presents a distributed sensor data aggregation platform in Rust demonstrating operating systems concepts including inter-process communication and concurrent data processing. The system employs a process-based architecture with ten child processes: five sensor readers connected via anonymous pipes, four aggregation workers sharing a named FIFO pipe, and one web server. A collector process groups sensor readings into fixed-duration time windows forwarded through POSIX shared memory with atomic counters. The aggregation engine applies Welford's online variance algorithm to compute streaming statistics; minimum, maximum, mean, and standard deviation; without a second data pass. Anomaly

DOI: 10.4018/979-8-2600-1101-0.ch004

detection uses a configurable z-score threshold per sensor type. Five benchmark trials confirm zero data loss across 698 readings, a throughput of 71.0 requests per second, a p99 latency of 14.08 milliseconds, and peak buffer utilisation of 0.01%, validating the platform's reliability guarantees under the intended workload.

1. INTRODUCTION

1.1 Problem Statement and Motivation

The reason and the motivation of designing the system of this chapter in this way comes from a common issue in modern sensor monitoring platforms. Nowadays, we found out that many sensor systems need to collect and aggregate readings from multiple physical devices at the same time (Henning & Hasselbring, 2019), but these devices do not always operate at the same rate or under the same timing conditions. Because of that, we suggest that this is not only a data processing problem, but also an operating systems problem (Alshehri & Bajaber, 2024). In the system of our chapter, the sensors may produce some readings around 10 to 20 Hz, while downstream stages such as aggregation, disk persistence, and web serving may execute at slower or less predictable rates. We suggest that this would happen because of the scheduling delays, lock contention, cache effects, and I/O operations. So, even though the pipeline may look stable at first, but there is still a mismatch between the fast producers and the slower consumers.

This mismatch becomes more and more serious and dangerous when the upstream side only has a small internal buffer. In our chapter, each simulated sensor has a 128-slot circular ring buffer (Nikolaev & Ravindran, 2022). If the reader process does not drain this buffer quickly enough, newly arriving readings will overwrite the oldest ones. As a result, the system may lose data without producing an explicit error signal. This is not what we want, it is not only a theoretical problem. At 20 Hz, a sensor may produce one reading at every 50 ms, so a 128-slot buffer can be filled in about 6.4 seconds if the consumer is delayed, the effect of this is tremendous. In other words, even a short disturbance, such as OS scheduling jitter or a disk flush that temporarily slows the pipeline, may already be enough to cause this failure mode.

For this reason, an intermediate buffer is needed between the sensor readers and the downstream processing stages. This buffer acts like a shock absorber. It separates the relatively fast and regular behaviour of sensor ingestion from the slower and more variable behaviour of aggregation, storage, and as well as the web service components. By doing this, the whole pipeline becomes more stable and less sensitive to short-term slowdowns. So, the main design question is not whether