

**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING**



**REPORT
MULTIDISCIPLINARY PROJECT (CO3109)**

**DEPLOYMENT OF AN ENVIRONMENTAL
MONITORING SYSTEM UTILIZING
REDPANDA DATA STREAMING AND GAMPS
COMPRESSION ALGORITHM**

SUPERVISOR: DIEP THANH DANG, Ph.D.

-----o0o-----

STUDENT 1:	THIEU QUANG TUAN ANH	2153171 - CE
STUDENT 2:	TRAN BAO NGUYEN	2153637 - CS
STUDENT 3:	NGUYEN MINH TRIET	2153915 - CS
STUDENT 4:	LE QUOC TUAN	2153944 - CE

HO CHI MINH CITY – Dec/2024

Table of Contents

Chapter 1 Introduction.....	4
Chapter 2 Background	6
2.1 Data Streaming	6
2.1.1 Introduction to Data Streaming	6
2.1.2 Importance of Data Streaming	7
2.1.3 Core Concepts and Components	8
2.2 Redpanda Streaming Data Platform	10
2.2.1 Core Features	10
2.2.2 Architecture	11
2.2.3 Relevance to the Project	13
2.3 Data Compression.....	13
2.3.1 Introduction to Data Compression	13
2.3.2 Classification of Data Compression Techniques	14
2.3.3 Importance of Data Compression in Data Streaming.....	15
2.4 GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling 16	
2.4.1 Problem Statement.....	16
2.4.2 Objectives	17
2.4.3 Related Works	17
2.4.4 GAMPS Compression	17
Chapter 3 Requirement Analysis.....	24
3.1 Functional Requirements.....	24
3.1.1 Data Streaming	24

3.1.2	Data Compression	24
3.2	Non-Functional Requirements	24
3.2.1	Redpanda Data Streaming	24
3.2.2	GAMPS Compression	25
Chapter 4	System Design and Implementation	26
4.1	System Conceptual View	26
4.2	System Logical View	27
4.2.1	IoTGateway Class	28
4.2.2	Server Class	29
4.2.3	MultisensorDataGrouper Class	30
4.3	System Process View	31
4.4	Redpanda Configuration	33
4.4.1	Global Configurations	34
4.4.2	Broker-Specific Configurations	36
4.4.3	Networking Considerations	38
4.4.4	Console Configuration	41
4.4.5	Summary	44
Chapter 5	User Manual	45
5.1	Prerequisites	45
5.2	Installation	45
5.2.1	Clone the Repository	45
5.2.2	Install Python Dependencies	45
5.2.3	Add the Dataset	45
5.3	Running the System	46

5.3.1	Start Redpanda.....	46
5.3.2	Run the Client.....	46
5.3.3	Run the Server	47
5.3.4	(Optional) Server Data Compression	47
5.3.5	(Optional) Server Data Compression	47
5.3.6	Stop Redpanda.....	47
Chapter 6 Project Evaluation and Demonstration		48
6.1	Redpanda Data Streaming.....	48
6.1.1	User Manual Demonstration	48
6.1.2	Requirement Evaluation	54
6.2	GAMPS Compression	59
6.2.1	Compression Performance	59
6.2.2	Scaling with Group Size.....	61
6.2.3	Compression Gain of Grouping	61
Chapter 7 Conclusion		65
Bibliography.....		66

Chapter 1

Introduction

In the modern era of Internet of Things (IoT), the growing number of interconnected devices generates an immense volume of data in real time. This data, collected from various sensors and devices, is crucial for enabling applications such as healthcare monitoring, industrial automation, and environmental tracking. However, managing, transmitting, and storing this data efficiently presents significant challenges. This is where data streaming and data compression become essential components of IoT systems.

Data streaming refers to the continuous transfer of data from devices to central systems for real-time analysis and decision-making. It enables IoT applications to process information as it is generated, allowing for immediate responses to critical events. For example, real-time traffic monitoring systems rely on streaming to dynamically adjust signals and reduce congestion. Without efficient data streaming, IoT systems risk delays, bottlenecks, and potential data loss, which could undermine the reliability and effectiveness of applications.

Beside with data streaming, data compression is the process of reducing the size of data while retaining its essential features. IoT devices often operate on limited bandwidth and storage, making compression critical for ensuring that large amounts of data can be transmitted and stored efficiently. Compression not only reduces network load and storage costs but also accelerates data transmission, making it possible to meet the stringent real-time requirements of IoT applications. For example, in remote monitoring systems, compression ensures that sensor data can be transmitted over low-bandwidth networks without sacrificing accuracy.

Using Data Streaming and Data Compression Together.

The combination of data streaming and compression addresses two critical aspects of IoT systems: handling high data velocity and managing large data volumes. Streaming ensures that data flows smoothly and in real time, while compression reduces the amount of data being stored, optimizing storage resources and network bandwidth while transmitting.

By enabling real-time data processing and efficient resource utilization, these technologies play a vital role in addressing the challenges raised by the evolving IoT landscape. Together, these technologies enable IoT applications to scale efficiently, reduce costs, and maintain high performance. This report studies these concepts, exploring their importance and the strategies for implementing them effectively in IoT ecosystems.

Chapter 2

Background

2.1 Data Streaming

2.1.1 Introduction to Data Streaming

Data streaming refers to the continuous flow of data generated in real-time from various sources. Unlike traditional batch processing, where data is collected and analyzed after accumulation, data streaming allows immediate processing and analysis as the data arrives. This method is essential for managing and utilizing high-velocity and high-volume data effectively.

For instance, consider social media platforms where user activities such as posts, comments, and likes generate a constant stream of data. Another example is the Internet of Things (IoT), where sensors in devices like smart thermostats, wearable health trackers, and autonomous vehicles produce streams of real-time data. Financial markets also rely on data streaming to process stock trades, market feeds, and economic indicators instantaneously. These scenarios demonstrate how streaming data powers critical real-time decision-making processes.

According to the website of AWS [1], a data stream is defined by several specific characteristics that set it apart from other forms of data processing:

- **Chronologically significant:** Each element in a data stream is associated with a timestamp, often making the timing of the data critical. The relevance of the data may diminish after a certain time frame. For instance, an application offering restaurant recommendations based on a user's current location must process geolocation data instantly, as its value declines if delayed.
- **Continuously flowing:** A data stream operates without a defined beginning or end, continuously collecting data for as long as needed. For example, server activity logs are generated and recorded as long as the server remains active.

- **Unique:** Data streams are challenging to retransmit due to their time-sensitive nature. As a result, real-time processing is crucial to ensure accuracy, as most streaming sources provide limited mechanisms for retransmission of missed data.
- **Nonhomogeneous:** Data streams can come from multiple sources with varying formats, such as JSON, Avro, or CSV, and include diverse data types like strings, numbers, dates, and binary data. Stream processing systems must be equipped to handle these variations efficiently.
- **Imperfect:** Streaming data may encounter issues like missing or corrupted elements due to temporary errors at the source. Ensuring data consistency can be difficult because of the continuous nature of streams. Processing and analytics systems typically implement data validation mechanisms to detect and address such errors effectively.

2.1.2 Importance of Data Streaming

The exponential growth of connected devices and the Internet of Things (IoT) has significantly increased the demand for data streaming. In 2024, approximately 18.8 billion IoT devices are connected globally, generating vast amounts of data daily [2].

This surge is driven by the proliferation of smart homes, industrial machinery, and other connected technologies that produce continuous data streams requiring immediate processing to yield actionable insights. Consequently, advancements in data collection and processing technologies have become essential to manage the immense volume and velocity of data generated by these devices.

Big data technologies have further propelled the importance of data streaming. Frameworks like Apache Kafka, Apache Flink, and Amazon Kinesis have emerged as powerful tools to manage these vast data streams, ensuring scalability, fault tolerance, and low-latency processing.

Modern applications demand real-time or near-real-time data processing to stay competitive and meet user expectations. For example, in e-commerce, real-time analytics help personalize customer experiences by recommending products based on browsing history and

preferences. Similarly, in healthcare, streaming data from wearable devices can alert doctors to anomalies in a patient's vital signs, enabling timely interventions.

The demand for real-time processing also extends to industries like telecommunications, where predictive maintenance relies on analyzing sensor data streams from network infrastructure. Such use cases highlight how data streaming ensures that businesses and systems can adapt and respond almost instantaneously, which is crucial in a fast-paced digital world.

2.1.3 Core Concepts and Components

The core concepts and components of data streaming provide the foundation for designing efficient systems that process and analyze continuous data flows. This section explores the essential elements of data streaming, including stream processing frameworks, producers and consumers, topics and partitions, and the differences between single-thread and multi-thread processing. The core components of data streaming are illustrated in Figure 2.1.

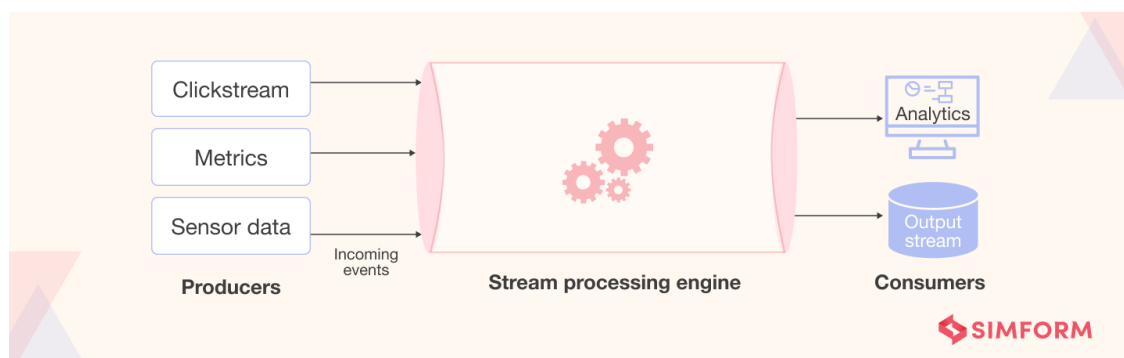


Figure 2.1 - Core Components of Data Streaming [3]

2.1.3.1 Stream Processing Frameworks

Stream processing frameworks serve as the backbone of data streaming systems. They provide the tools and infrastructure to process and analyze continuous data streams in real-time. Examples include **Redpanda**, **Apache Kafka**, and **Apache Flink**, each offering distinct capabilities like scalability, fault tolerance, and low-latency processing.

- **Redpanda** simplifies streaming by providing a Kafka-compatible API with faster performance and reduced resource requirements.

- **Apache Flink** is designed for stateful computations and complex event processing in real-time. These frameworks ensure data integrity, manage workloads, and allow seamless integration with existing systems.

2.1.3.2 Producers and Consumers

Producers and consumers are fundamental building blocks of data streaming systems:

- **Producers:** Generate and send data to a streaming platform. For example, IoT devices, web applications, or server logs can act as producers.
- **Consumers:** Receive, process, and analyze the data from the streaming platform. They use the data for various purposes, such as generating insights, compressing data or triggering automated actions.

Producers and consumers communicate through a shared system, enabling the decoupling of data generation from data processing. This architecture ensures scalability and flexibility.

2.1.3.3 Topics and Partitions

Topics and partitions are mechanisms used to organize and manage data within a stream processing framework:

- **Topics:** Logical channels or categories to which producers send data and consumers subscribe to receive data. For instance, a topic might represent sensor readings from a specific location.
- **Partitions:** Subdivisions within a topic, enabling parallelism by distributing data across multiple nodes or servers. Each partition contains an ordered sequence of records, facilitating efficient load balancing and scaling.

Partitions are critical for achieving high throughput and ensuring the system can handle large-scale data streams.

2.1.3.4 Single-thread vs. Multi-thread Processing

The choice between single-thread and multi-thread processing significantly impacts the performance and efficiency of a streaming system:

- **Single-thread Processing:** Involves sequential handling of data streams. It is simpler to implement but may struggle to process high-throughput data streams due to limited concurrency.
- **Multi-thread Processing:** Distributes tasks across multiple threads, enabling parallel processing and improved performance. It is ideal for high-throughput scenarios but introduces complexity in managing thread safety and synchronization.

Multi-thread processing is typically preferred for modern streaming systems due to its ability to handle the massive data volumes generated by IoT and other real-time applications.

2.2 Redpanda Streaming Data Platform

Redpanda is a modern streaming data platform designed to handle high-throughput, low-latency workloads with simplicity and efficiency. As a Kafka-compatible alternative, Redpanda provides a robust and scalable architecture tailored to real-time data processing. This section discusses its core features, architecture, and advantages, making it an ideal choice for streaming applications.



Figure 2.2 - Redpanda's Logo [4]

2.2.1 Core Features

According to Redpanda Website [5], Redpanda stands out as a high-performance, cost-effective, and easy-to-use streaming data platform, specifically designed to address the challenges of modern data streaming workloads. Its unique features make it different from other streaming data platforms:

- **Thread-per-core Architecture:** Redpanda, built in C++ with thread-per-core architecture, maximizes performance with up to 10x lower latencies.
- **Single binary:** Redpanda nodes are self-contained, with built-in schema registry, HTTP proxy, and Kafka-compatible APIs, managed using Raft. They require no external dependencies like JVM or ZooKeeper, ensuring faster boot times, simpler CI/CD, and more reliable production setups.

- **Native Raft:** Redpanda uses the Raft consensus protocol for efficient data management, ensuring performance, safety, and reliability across workloads, without the need for additional quorum servers.
- **Data sovereignty:** Redpanda Cloud's BYOC model combines self-hosted control with cloud convenience, offering fully managed clusters within your VPC. Redpanda handles provisioning, monitoring, and upgrades, while keeping your data and credentials securely within your environment.

2.2.2 Architecture

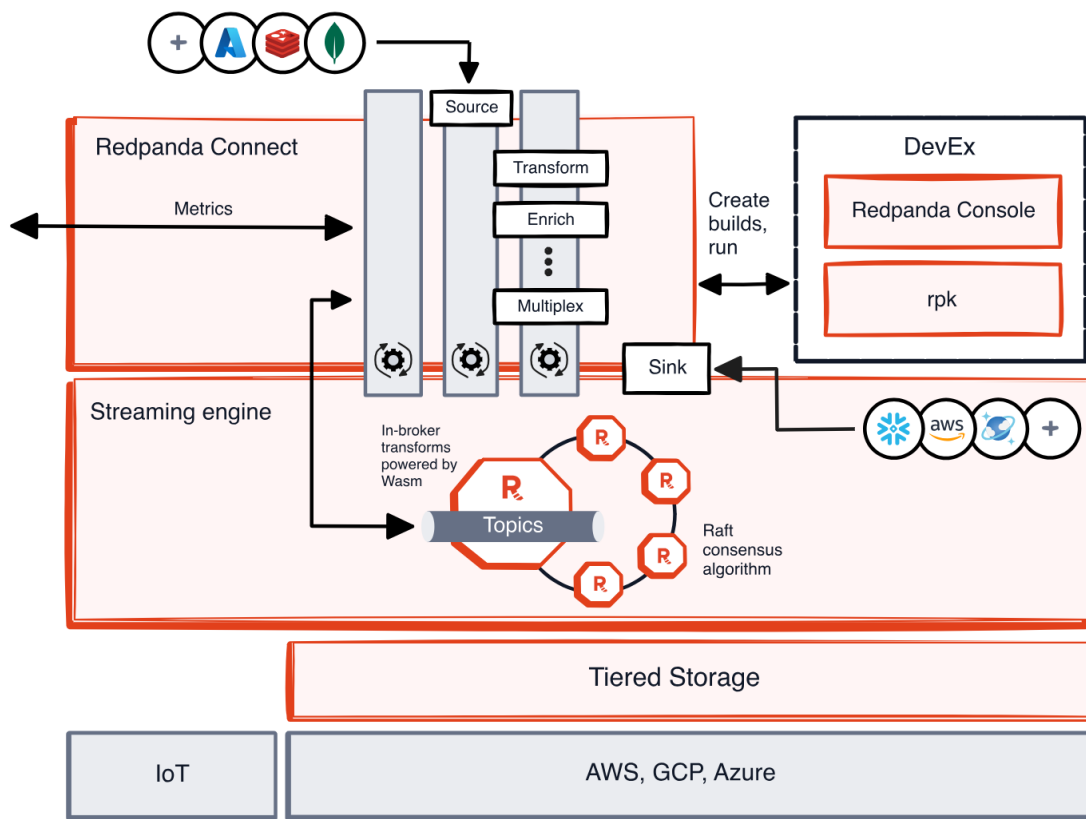


Figure 2.3 - Redpanda's Architecture [5]

Redpanda's architecture is designed to deliver high performance, reliability, and simplicity for real-time data streaming. The following are its core components and their functionalities, as illustrated in the architecture diagram in Figure 2.3:

- **Redpanda Connect:**

Redpanda Connect bridges external systems and the Redpanda streaming platform. It provides:

- *Sources and Sinks*: Connectors to ingest data from external sources and deliver processed data to sinks like databases or cloud storage.
- *Data Transformation and Enrichment*: Built-in mechanisms for transforming, enriching, and multiplexing data in real time, powered by WebAssembly (Wasm) for in-broker transformations.
- *Metrics*: Tools to monitor and analyze data pipeline performance.

This component simplifies the integration of Redpanda into diverse ecosystems, supporting a wide range of data workflows.

- **Streaming Engine:**

At the core of Redpanda is its high-performance streaming engine, responsible for managing data topics and ensuring reliable message processing. Key features include:

- *Topics*: The primary unit for organizing messages within Redpanda. Topics are distributed across partitions to enable scalability and parallel processing.
- *Raft Consensus Algorithm*: Ensures reliable data replication and fault tolerance by synchronizing data across nodes, even in cases of partial failure. This design guarantees data consistency and cluster reliability under high loads.

- **Developer Experience (DevEx):**

Redpanda enhances usability through developer-friendly tools:

- *Redpanda Console*: A web-based interface for monitoring, managing, and configuring clusters.
- *rpk (Redpanda CLI)*: A command-line tool for quick cluster setup, management, and debugging.

These tools make it easier for developers to deploy, monitor, and manage Redpanda clusters efficiently.

- **Tiered Storage:**

Redpanda offers scalable and cost-efficient data storage through **Tiered Storage**, which allows log segments to be offloaded to object storage in near real time. Key benefits include:

- Long-term data retention.
- Topic recovery without sacrificing performance.

- Seamless integration with cloud storage services like AWS S3, Google Cloud Platform, and Azure
- **Cloud and IoT Integration:**
 - Redpanda supports diverse deployment environments, including:
 - *IoT Workloads:* Handles data from edge devices in real-time streaming use cases.
 - *Cloud Deployments:* Fully compatible with cloud-native platforms, enabling integration with services such as Snowflake, AWS, and more.

This architecture emphasizes Redpanda's ability to handle real-time data streaming workloads with high performance, reliability, and adaptability to various use cases, from IoT to enterprise cloud systems.

2.2.3 Relevance to the Project

In this project, Redpanda serves as the core streaming platform, efficiently managing real-time environmental monitoring data. Its high performance and simplified architecture enable seamless integration with existing systems, ensuring reliable and scalable data processing. The platform's cost efficiency and operational simplicity make it an ideal choice for handling continuous data streams in a resource-effective manner.

By leveraging Redpanda, the project benefits from a robust and developer-friendly streaming data platform that aligns with the requirements of modern, real-time data processing applications.

2.3 Data Compression

2.3.1 Introduction to Data Compression

Data compression involves encoding information using fewer bits than the original representation. The goal is to reduce the storage space required and improve data transmission efficiency. Compression is essential in systems where bandwidth, storage, and processing capabilities are limited or costly.

For instance, in streaming systems, data compression helps optimize the transfer of continuous streams, reducing latency and improving throughput without significantly affecting data quality.

2.3.2 Classification of Data Compression Techniques

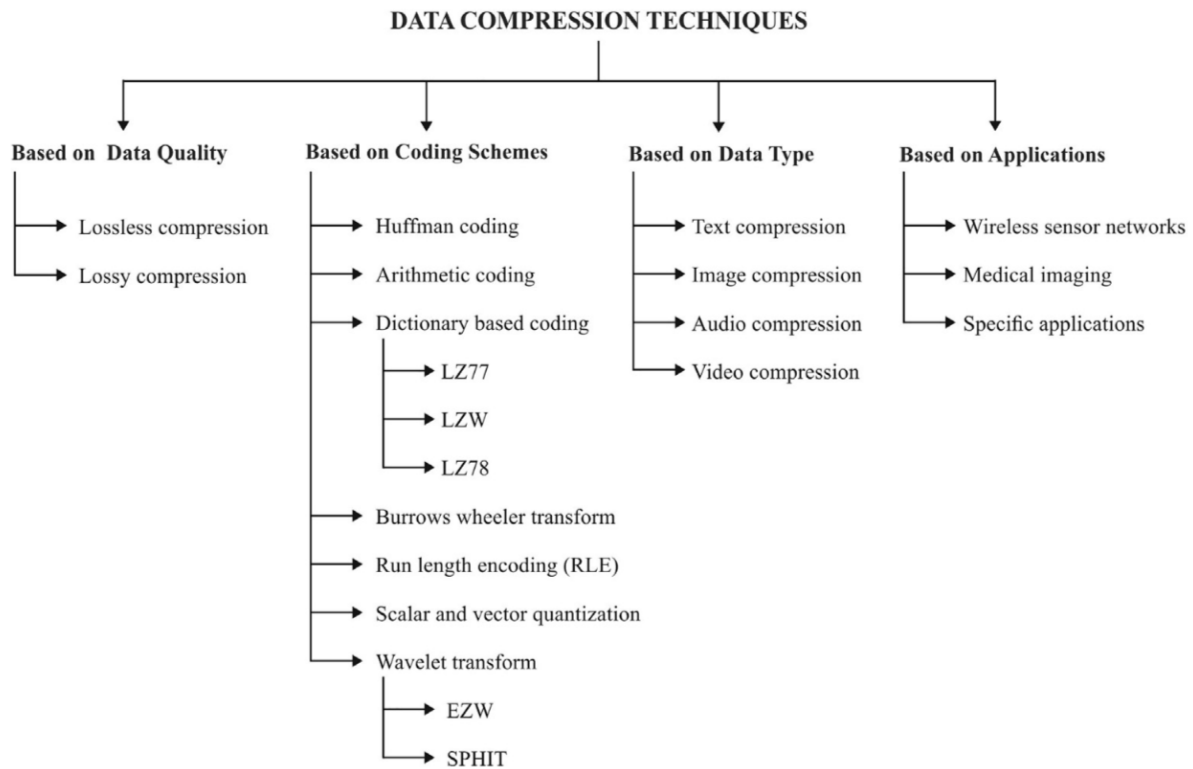


Figure 2.4 - Classification of DC techniques [4]

Data compression techniques can be broadly categorized based on specific criteria, as illustrated in Figure 2.4. The key categorizations are as follows:

1. Based on Data Quality

- **Lossless Compression:** Ensures that the original data can be perfectly reconstructed after compression. This is suitable for text and binary files where data integrity is critical.
- **Lossy Compression:** Reduces file size by discarding less significant data, commonly used for images, audio, and video, where slight data loss is acceptable for significant size reduction.

2. Based on Coding Schemes

- **Huffman Coding:** A variable-length coding technique based on symbol frequencies.
- **Arithmetic Coding:** Encodes entire data streams as a single number.
- **Dictionary-Based Coding:** Includes algorithms like LZ77, LZW, and LZ78, which replace repetitive patterns with shorter representations.
- **Run-Length Encoding (RLE):** Compresses data by storing repeated values as a single count and value pair.
- **Wavelet Transform:** A lossy technique used in modern image and video compression (e.g., EZW and SPIHT).

3. Based on Data Type

- **Text Compression:** Techniques designed for reducing the size of text files.
- **Image Compression:** Focuses on reducing the size of image files while retaining visual quality (e.g., JPEG).
- **Audio Compression:** Applied to sound data, balancing quality and size (e.g., MP3).
- **Video Compression:** Reduces the size of video files by compressing frames (e.g., H.264).

4. Based on Applications

- **Wireless Sensor Networks:** Optimized for low-power, high-throughput scenarios.
- **Medical Imaging:** Ensures data integrity while reducing storage requirements for medical scans.
- **Specific Applications:** Custom techniques designed for domain-specific needs.

2.3.3 Importance of Data Compression in Data Streaming

Data compression plays a critical role in modern data streaming systems, where vast amounts of data are continuously generated, transmitted, and processed in real time. Its importance can be understood through the following key aspects:

- **Efficient Bandwidth Utilization:** Compression reduces data size, allowing more data to be transmitted over limited bandwidth efficiently.
- **Reduced Latency:** Smaller data packets enable faster transmission, ensuring real-time responsiveness in streaming systems.
- **Cost Efficiency:** By compressing data, storage requirements are minimized, lowering storage costs significantly.
- **Improved Scalability:** Compression enables systems to handle growing data volumes without overwhelming infrastructure.
- **Resource Efficiency:** Compression minimizes the processing and transmission burden on resource-constrained devices, such as IoT sensors.
- **Enhanced Analytics:** Compressed data is faster to process, supporting real-time insights and advanced analytics.

In this project, the GAMPS compression algorithm is used to reduce data storage requirements, optimizing the handling of large environmental datasets while maintaining cost efficiency.

2.4 GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling

2.4.1 Problem Statement

The rapid proliferation of wireless sensor networks (WSNs) and Internet of Things (IoT) devices has led to massive streams of time-series data from distributed sensors. This data, while rich in information, poses challenges in terms of storage, transmission bandwidth, and energy efficiency. Traditional compression methods fail to adequately exploit spatial and temporal correlations between sensors, especially in systems where data is both voluminous and redundant. The GAMPS (Grouping and Amplitude Scaling) framework addresses these

issues by introducing a novel, scalable approach to compressing multi-sensor data while preserving critical information [7].

2.4.2 Objectives

The main goal of GAMPS is to minimize the memory and transmission cost associated with storing and sharing sensor data by:

1. Exploiting correlations between sensor data streams through **grouping**.
2. Using **amplitude scaling** to align data streams for more efficient representation.
3. Maintaining fidelity and accuracy within acceptable bounds by employing relaxed error metrics.

2.4.3 Related Works

- **Dimensionality Reduction Techniques:** Methods like Principal Component Analysis (PCA) and Singular Value Decomposition (SVD) are widely used for compressing multi-dimensional data but often lack adaptability to dynamic sensor environments. These techniques focus on global data properties rather than localized correlations exploited in GAMPS.
- **Wavelet and Symbolic Transformations:** Wavelet-based compression is effective for reducing redundancy in time-series data. However, it may fail to leverage inter-sensor correlations, which GAMPS addresses by grouping signals with similar behavior.
- **Distributed Source Coding:** Slepian-Wolf coding optimize data compression in sensor networks by sharing information among sensors. GAMPS builds on similar principles but integrates amplitude scaling for higher compression efficiency.
- **Set Cover-Based Approximation:** Weighted set cover problems have been explored in computational frameworks for resource optimization. GAMPS adapts this approach to identify optimal groupings and scaling factors for multi-sensor data streams.

2.4.4 GAMPS Compression

2.4.4.1 Approximation of a Single Signal

A time series can be compressed using the bucketing scheme from figure below, which groups consecutive data points into buckets. The algorithm adds points to a bucket until the range of values within it exceeds 2 epsilon, after which a new bucket is created. Each bucket's values are approximated within an error of epsilon using the average of its maximum and minimum. Graphically, this corresponds to bounding the bucket's data within a rectangle of height 2 epsilon. This greedy method ensures an optimal number of buckets.

For k signals, additional compression is achieved by sharing portions of buckets. By relaxing the error tolerance to 3 epsilon, the algorithm identifies a minimal set of shared rectangular boxes to cover all signals, modeled as a set-cover problem.

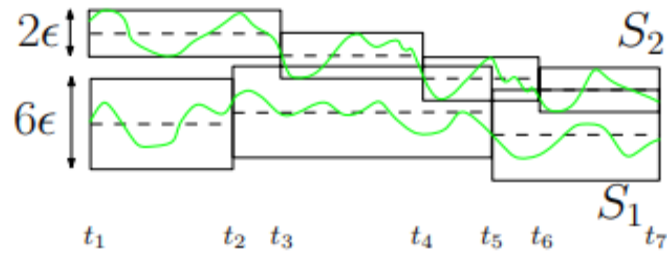


Figure 2.5 - Between time instants t_3 and t_7 , the relaxed approximation segment bounding rectangles for signal s_1 intersects the approximation segment bounding rectangles of s_2 [7]

2.4.4.2 Compression by Interval Sharing

The bucket representation of a signal divides it into rectangular segments called buckets, each containing a range of consecutive data points. When the range of values within a bucket exceeds the predefined limit, a new bucket is created. This ensures that all points in a bucket can be approximated within a fixed error tolerance.

After applying this method to multiple signals, the goal is to reduce storage further by sharing portions of buckets across signals. This involves relaxing the error tolerance and identifying intervals where one signal's approximation can represent others. Relaxed bounding boxes, which expand the original limits, are used to check if one signal fully covers another within the same interval. If so, shared storage can be achieved with minimal error.

Weights are assigned to each interval, representing the cost of using one signal's bucket to approximate others in the same range. The total storage cost is minimized using a weighted

set cover algorithm, which identifies the best combination of shared buckets. This approach ensures efficient compression while maintaining acceptable approximation accuracy.

2.4.4.3 Amplitude Scaling

One way to transform correlated signals for compression is by using linear regression. When two signals are correlated, one can often be approximated by a linear function of the other, where the relationship between the signals is described by two coefficients, which we calculate to minimize a chosen error metric. These coefficients allow us to reconstruct one signal using the other, thus enabling compression.

However, this approach becomes complicated when we want to minimize the maximum error (L^∞ error) instead of the average error (L^2 error), as the required computation of the coefficients for each segment of the signals is complex and costly. A more efficient alternative is to transform the signals individually. Instead of applying the same transformation parameters across an entire signal or segment, we can calculate a ratio between the corresponding values of the two signals at each point in time. This ratio captures the relative values between the two signals, allowing us to represent one signal in terms of the other. This method reduces the complexity and computation cost, as it avoids the need for segmenting the signals and calculating parameters for each segment.

Using Ratio Signals

When it comes to signal compression, there are various approaches to transforming signals, such as using ratio or delta signals. A ratio signal involves calculating the ratio between corresponding values of two signals at each time point, which can lead to a smoother and more compressible representation. This is particularly useful when the relationship between the signals is approximately constant over time, making the ratio relatively flat.

For instance, in scenarios where multiple sensors are deployed in a physical environment, the relationship between sensor readings can often be described by a function, even if it is nonlinear. Over short time intervals, this relationship can still be approximated linearly, and the ratio between sensor readings may remain stable. This is especially true when one sensor's value changes in proportion to another's, such as in the case of temperature

sensors in a data center, where the temperature ratio between two sensors remains relatively constant even as the sensors' values increase.

In contrast, a delta signal, which measures the difference between two signals, tends to be more volatile and less compressible, as the differences can vary significantly over time. However, in some cases, delta signals might be useful, particularly when sensors operate independently or do not share a physical relationship.

Empirical tests have shown that ratio signals are generally more compressible than delta signals for real-world sensor data. The smoother nature of ratio signals allows for better compression ratios, making them more space-efficient for maintaining a given error bound. Therefore, for the majority of cases, using ratio signals is preferred, although delta signals may still be appropriate in specific contexts.

This framework provides the flexibility to use either signal type, depending on the nature of the sensors and the data they produce.

Compressing Base and Ratio Signals

Motivated by the earlier observation, the approach separates the compression of the base signal and the ratio signal between two signals. By compressing these signals separately, the signal S_i can later be reconstructed with a guaranteed maximum error. To achieve this, the piecewise constant bucket approximation is used on both the base signal and the ratio signal, resulting in error bounds for each signal.

The main challenge is ensuring that the total error during reconstruction remains within the desired limit. The relationship between the errors of the base signal and the ratio signal is essential in determining the final maximum reconstruction error.

The procedure is part of the GAMPS framework, where the first step is grouping similar sensor signals together. Within each group, one signal is selected as the base, and the others are expressed in terms of their ratio relative to the base signal. This grouping is critical for optimizing compression. Once grouped, the ratio signals of all signals in a group tend to be highly compressible, which makes them suitable for the bucketing approximation and interval-sharing techniques.

In practice, these ratio signals are often so sparse that interval sharing is not even necessary, simplifying the overall compression process. This methodology is demonstrated using data from a data center, where sensor signals are grouped and compressed efficiently.

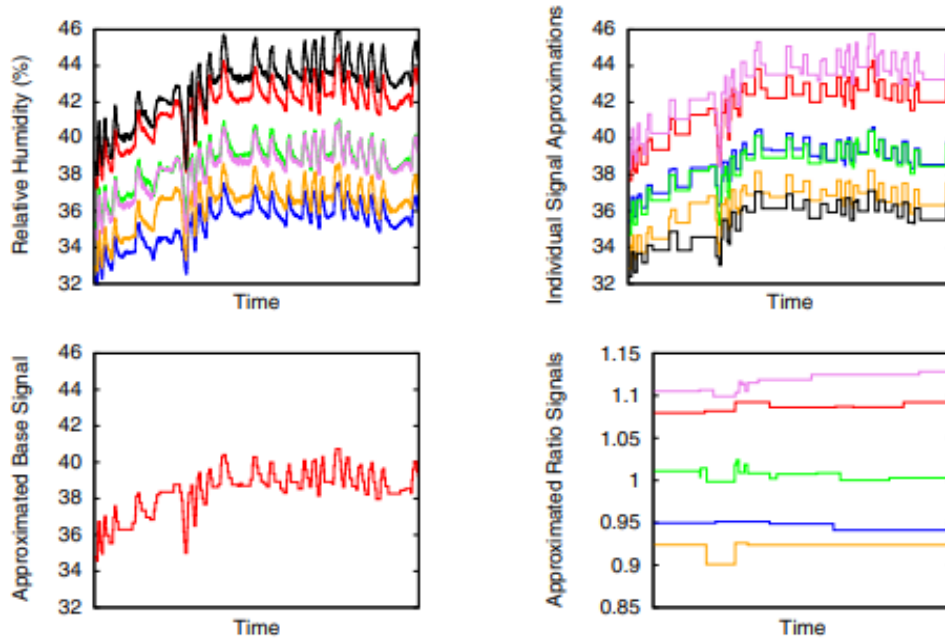


Figure 2.6 - The top row shows the humidity data and the result of individual signal compression. The bottom row shows the bucketing compression of a base signal, and the ratio signals of the other 5 signals. In all cases, the maximum allowed L_∞ error is 1% [7]

2.4.4.4 Implementation Detail

The algorithm is implemented as proposed method in the publication. Figure 2.7 is the pseudo code for implementation:

Algorithm 1 STATGROUP(\mathcal{S}, ε)

```

1:  $C = \emptyset, W = \emptyset$ 
2:  $S' = S$ 
3: while  $S' \neq \emptyset$  do
4:   Set  $\varepsilon_1 = 0.4\varepsilon$  and determine  $\varepsilon_2$  using  $\varepsilon_1$ .
5:   Pick one signal from  $S'$ , call it  $S_j$ .
6:   Let  $c_j = \text{Bucket}(S_j, \varepsilon_1)$ .
7:    $C = C \cup \{c_j\}$ 
8:   for all  $S_i$  in  $S$  do
9:     Take  $S_j$  as the base signal, and compute ratio signal
        $R(S_i, S_j)$ .
10:    Let  $w(i, j) = \text{Bucket}(R(S_i, S_j), \varepsilon_2)$ .
11:     $W = W \cup \{w(i, j)\}$ 
12:   end for
13:    $S' = S' \setminus \{S_i\}$ 
14: end while
15: Facility-Location( $C, W$ )
16: Return the total cost of setting up the facilities and serving
    clients.
```

Figure 2.7 - Pseudo Code Implementation for Static Grouping [7]

2.4.4.5 Dynamic Grouping

The physical environment changes over time, which means a signal grouping that is optimal at one moment may not remain so in the future. This is particularly important for long-term data archiving, such as monitoring humidity and temperature in large data centers, where trends can shift. To address this, we implemented a dynamic version of our grouping scheme, detailed in Algorithm 2.

The Dynamic Grouping (DynGroup) heuristic starts by defining a time window and computing an initial set of groups using the StatGroup algorithm. As new data becomes available, the groups are recomputed for each new time window, and the window size is dynamically adjusted to better align with changes in the data. The algorithm decides whether to double, halve, or keep the current window size based on which option provides the best memory performance.

In this approach, the current group size is used to process the data. A function called ConstructSet is used to create a subset of the dataset, starting from a specific time point and including a certain number of data samples across all time series. Initially, the algorithm compares the memory usage of maintaining the current group size versus halving it. If halving improves memory efficiency, the group size is reduced. If not, the algorithm checks whether doubling the group size results in better memory savings. If doubling is more efficient, the

group size is increased; otherwise, it remains unchanged. This process is repeated for each batch of data, with the batch size equal to the current group size. The remaining steps in the algorithm are straightforward to follow.

Algorithm 2 DYN_{GROUP}(S, ε)

```

1:  $wsiz e = 100, lastpt = 0$ 
2: while  $lastpt \leq tsize$  do
3:    $S_{11} = ConstructSet(S, lastpt, wsiz e)$ 
4:    $S_{21} = ConstructSet(S, lastpt, wsiz e/2)$ 
5:    $S_{22} = ConstructSet(S, lastpt + wsiz e/2, wsiz e/2)$ 
6:    $m_{11} = StatGroup(S_{11}, \varepsilon), m_{21} = StatGroup(S_{21}, \varepsilon), m_{22}$ 
      $= StatGroup(S_{22}, \varepsilon)$ 
7:   if  $m_{11} \geq m_{21} + m_{22}$  then
8:      $lastpt += wsiz e/2$ 
9:      $wsiz e = wsiz e/2$ 
10:  else
11:     $S_{12} = ConstructSet(S, lastpt + wsiz e, wsiz e)$ 
12:     $S_3 = ConstructSet(S, lastpt, wsiz e * 2)$ 
13:     $m_{12} = StatGroup(S_{12}, \varepsilon)$ 
14:     $m_3 = StatGroup(S_3, \varepsilon)$ 
15:    if  $m_{11} + m_{12} \geq m_3$  then
16:       $lastpt += wsiz e * 2$ 
17:       $wsiz e = wsiz e * 2$ 
18:    else
19:       $lastpt += wsiz e$ 
20:    end if
21:  end if
22:  Update the index structure with the chosen representa-
     tions in the last iteration.
23: end while

```

Figure 2.8 - Pseudo Code Implementation for Dynamic Grouping [7]

Chapter 3

Requirement Analysis

3.1 Functional Requirements

3.1.1 Data Streaming

- Deploy and configure the Redpanda platform as the core streaming engine to handle real-time environmental data from an IoT Gateway.
- Stream data from the IoT Gateway to Redpanda topics for efficient real-time processing.

3.1.2 Data Compression

Apply the GAMPS compression algorithm on the server to reduce the size of environmental sensor data, optimizing storage.

3.2 Non-Functional Requirements

3.2.1 Redpanda Data Streaming

1. **Performance:** Redpanda must handle the streaming of maximum 5KB of data from the IoT Gateway every 31 seconds, comprising humidity, temperature, light, and voltage values.
2. **Low Latency:** Provide low-latency streaming so that the 5KB data batches from the IoT Gateway are available in real time for the consumer (server) within under 1 second after production.
3. **Data Consistency:** Ensure strict ordering and consistency of the streamed data chunks, with no data loss or duplication during transmission and storage.
4. **Fault Tolerance:**
 - Handle potential disconnections or network failures between the IoT Gateway and Redpanda, ensuring data is buffered and retransmitted when the connection is restored.

- Handle potential disconnections or network failures between the Server and Redpanda, ensuring data is buffered and retransmitted when the connection is restored.

5. Monitoring and Manageability: Provide tools to monitor the streaming of data from the IoT Gateway to Redpanda and to the Server, allowing for real-time tracking of data throughput, latency, and system health.

3.2.2 GAMPS Compression

- 1. Compression Efficiency:** Achieve a compression scale factor greater than 6 while maintaining an error acceptance of 2% during the GAMPS compression process.
- 2. Grouping Efficiency:** Ensure that the grouping-based compression scale factor is at least 2x higher than distinct (individual) mapping, leveraging inter-signal correlations for improved compression.
- 3. Dynamic Grouping Advantage:** Ensure that the dynamic grouping compression factor is at least 2x higher than static grouping, adapting to changing environmental trends for optimized data representation.

Chapter 4

System Design and Implementation

4.1 System Conceptual View

This section provides a high-level overview of the IoT-based environmental monitoring system, emphasizing the integration of Redpanda for real-time data streaming and GAMPS for optimizing data storage. The primary objective is to deploy a system capable of efficiently handling environmental data through streaming, compression, and storage mechanisms. The sensor nodes will be simulated to focus on the core components: the Redpanda streaming platform and the server running the GAMPS compression algorithm.

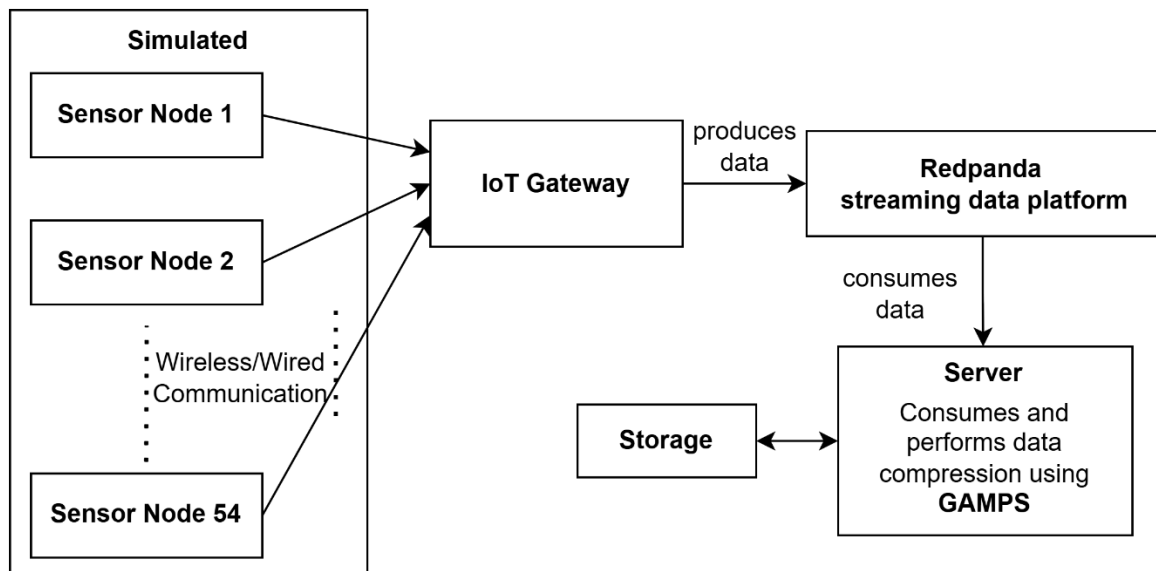


Figure 4.1 - System Architure Block Diagram

The system architecture is designed to ensure efficient data collection, streaming, compression, and storage for the IoT environmental monitoring application. The block diagram (Figure 4.1) illustrates the main components and their interactions:

1. *Sensor Nodes (Simulated)*: The system starts with multiple IoT sensor nodes (e.g., Sensor Node 1 to Sensor Node 54) that simulate the monitoring of laboratory environmental parameters, including temperature, humidity, light, and voltage. These

nodes communicate with the IoT Gateway through scripts that simulate wired or wireless communication protocols.

2. *IoT Gateway*: This gateway aggregates and preprocesses the data by attaching metadata such as timestamps and device identifiers. The gateway produces the preprocessed data to the Redpanda streaming platform.
3. *Redpanda Streaming Data Platform*: The Redpanda platform is the core of the system, acting as a high-performance data streaming engine. Producers (IoT Gateway) send data to Redpanda, and consumer applications (Server with GAMPS Compression) retrieve this data for processing.
4. *Server with GAMPS Compression*: The server consumes data from the Redpanda topic and applies the GAMPS compression algorithm. This step is critical for optimizing storage, as it reduces the size of the data while maintaining its integrity. The compressed data is then stored for long-term use, facilitating efficient data retrieval for analytics and reporting.

This architecture is designed to efficiently handle high-frequency environmental data, ensuring reliable data collection, compression, storage, and retrieval. By integrating Redpanda and GAMPS, the system provides a robust and scalable solution for IoT-based environmental monitoring.

4.2 System Logical View

The **Logical View** provides a detailed breakdown of the system's key components and their interactions, focusing on how Redpanda facilitates data streaming and how the server integrates with the GAMPS compression algorithm. This section describes the primary classes, their attributes, and methods, providing an understanding of the internal structure and logic of the system.

The class diagram (Figure 4.2) illustrates the structure and interaction of the key components in the system, outlining their logical structure:

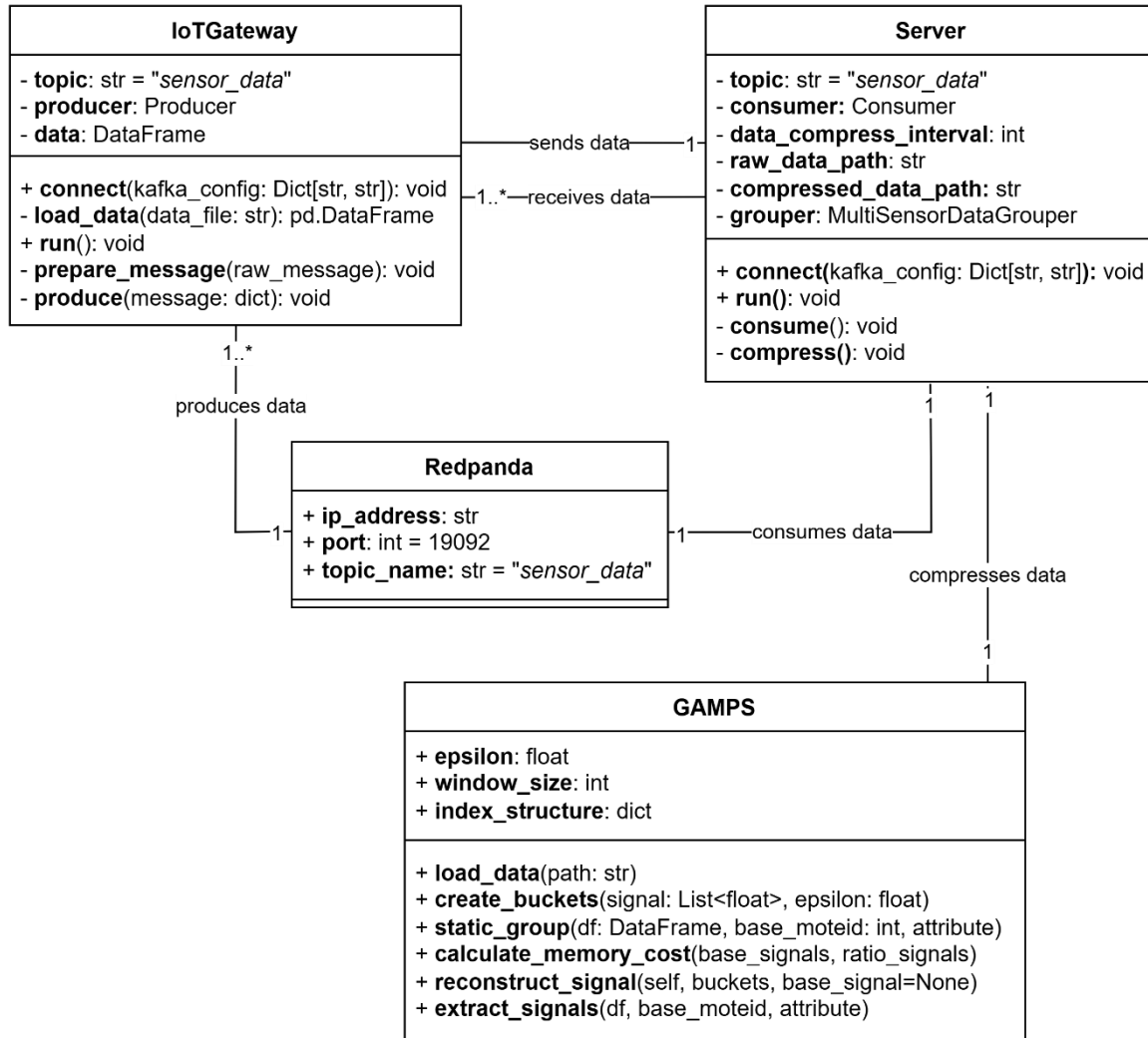


Figure 4.2 - Class Diagram

4.2.1 IoTGateway Class

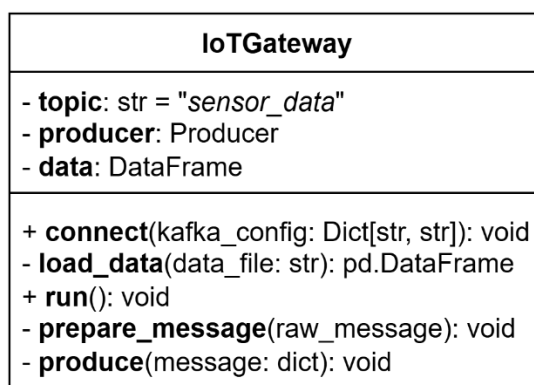


Figure 4.3 - IoTGateway Class

The **IoTGateway** class represents the IoT Gateway responsible for producing sensor data to the Redpanda platform. This class handles data preparation and streaming.

Attributes:

- **topic:** Specifies the topic name (**sensor_data**) to which the data will be produced.
- **producer:** Represents the Kafka producer instance, responsible for managing the data production process to the Redpanda streaming data platform.
- **data:** Holds the DataFrame containing the simulated sensor data to be sent to the streaming platform.

Methods:

- **connect(kafka_config):** Establishes a connection to the Redpanda streaming platform, configuring the producer for communication.
- **load_data(data_file):** Loads sensor data from a dataset file path into the **data** attribute, preparing it for further processing.
- **run():** starts running the simulation and producing data to Redpanda.
- **prepage_message(raw_message):** Formats raw messages into the correct format for seamless streaming to Redpanda.
- **produce(message):** Sends the prepared messages to the specified topic (**sensor_data**) on the Redpanda platform.

4.2.2 Server Class

Server
<ul style="list-style-type: none"> - topic: str = "sensor_data" - consumer: Consumer - data_compress_interval: int - raw_data_path: str - compressed_data_path: str - grouper: MultiSensorDataGrouper
<ul style="list-style-type: none"> + connect(kafka_config: Dict[str, str]): void + run(): void - consume(): void - compress(): void

Figure 4.4 - Server Class

The **Server** class is responsible for consuming real-time sensor data from the Redpanda streaming platform, managing data processing, and performing periodic compression using the GAMPS algorithm.

Attributes:

- **topic:** Specifies the topic name (sensor_data) from which data is consumed.
- **consumer:** Represents the Kafka consumer instance, which retrieves sensor data from Redpanda.
- **data_compress_interval:** Defines the time interval (in seconds) for compressing batches of raw data. Default is 5 hours.
- **raw_data_path:** The file path where raw sensor data is stored before compression.
- **compressed_data_path:** The directory path where compressed data is saved after processing.
- **grouper:** An instance of the **MultiSensorDataGrouper** class that handles the GAMPS compression algorithm for optimizing storage.

Methods:

- **connect(kafka_config):** Establishes a connection to the Redpanda (Kafka) broker and subscribes to the specified topic (**sensor_data**).
- **run():** Starts the server by running both the data consumption and compression processes concurrently using threads.
- **consume():** Retrieves data from the Redpanda platform and writes it to the raw data file for processing and storage.
- **compress():** Periodically compresses the raw data stored in the file using the GAMPS compression algorithm and saves the compressed output for long-term storage.

4.2.3 MultisensorDataGrouper Class

MultiSensorDataGrouper
+ epsilon : float + window_size : int + index_structure : dict
+ load_data (path: str) + create_buckets (signal: List<float>, epsilon: float) + static_group (df: DataFrame, base_moteid: int, attribute) + calculate_memory_cost (base_signals, ratio_signals) + reconstruct_signal (self, buckets, base_signal=None) + extract_signals (df, base_moteid, attribute)

Figure 4.5 – MultiSensorDataGrouper Class

The **MultiSensorDataGrouper** class is a key component in the GAMPS compression process, designed to efficiently compress and manage sensor data. This class leverages GAMPS algorithm (explained in the Background section) to reduce memory usage while retaining essential data features. It provides tools for data bucketing, signal extraction, and compression, which are integral to optimizing storage in the IoT environmental monitoring system.

The **MultiSensorDataGrouper** facilitates the core functionality of the GAMPS compression algorithm by grouping sensor data into a manageable format. Its capabilities include creating data buckets, calculating memory costs, and reconstructing signals from compressed formats.

4.3 System Process View

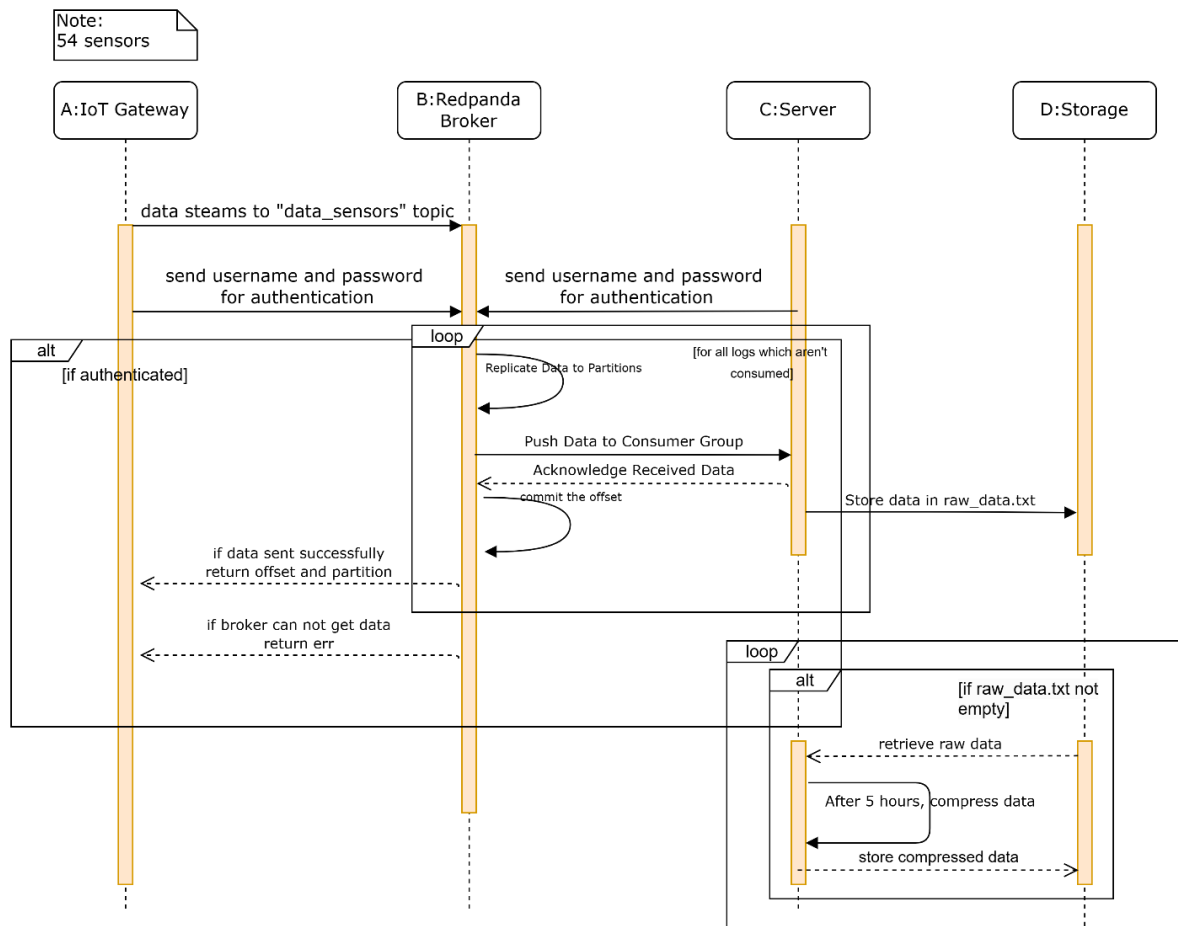


Figure 4.6 - Sequence Diagram

The **Process View** describes the flow of data through the system, outlining the sequence of interactions between the key components of the IoT environmental monitoring system: the IoT Gateway, Redpanda, Server, and Storage. This view emphasizes how data is generated, transmitted, processed, and stored in real time, ensuring efficient handling and long-term optimization.

The sequence diagram (Figure 4.6) illustrates the following process flow:

1. Sensor Data Generation:

- The system begins with 54 sensors generating environmental data (e.g., temperature, humidity).
- Sensors stream the data to the data_sensors topic on the Redpanda broker.

2. Authentication:

- Both sensors and the server send a username and password to the Redpanda broker for authentication.
- If authentication succeeds, the broker allows data streaming or retrieval. If it fails, an error is returned.

3. Data Streaming and Partitioning:

- Once authenticated, sensor data is pushed to the data_sensors topic.
- The Redpanda broker replicates the data into multiple partitions for fault tolerance and scalability.
- It acknowledges receipt of the data and commits the offset for reliable streaming.

4. Data Consumption by Server:

- The server, acting as a consumer, retrieves data from the Redpanda broker for all logs that haven't been consumed yet.
- It stores the raw data in a file (raw_data.txt) located in the server's storage.

5. Periodic Data Compression:

- Every 5 hours, the server checks if the raw_data.txt file is not empty.
- If data exists, the server retrieves the raw data and compresses it using the GAMPSS compression algorithm.
- The compressed data is then stored in a separate location for long-term storage optimization.

6. Acknowledgment and Error Handling:

- If data is successfully sent by the sensors or retrieved by the server, the broker returns the offset and partition details.
- If the broker encounters any issues, such as being unable to get data, it returns an error message to the sender.

4.4 Redpanda Configuration

This section provides an in-depth explanation of the configurations used in our Redpanda Docker-Compose setup. The configurations are divided into global configurations, broker-specific settings, networking considerations, and console configurations. Each subsection elaborates on the purpose, rationale, and overall contribution of these configurations to building our data streaming system.

4.4.1 Global Configurations

Global configurations are foundational settings that apply across all services in the Docker-Compose environment. These configurations establish the network infrastructure, manage storage durability, and optimize the runtime behavior of the system, ensuring seamless service interoperability and data resilience.

4.4.1.1 Networks

The network configuration in our system is crucial for enabling secure and efficient communication between services within an isolated virtual environment. It establishes a bridge network that facilitates interaction among Docker containers while keeping them isolated from the host system.

- **Configuration:**

```
yaml
networks:
  redpanda_network:
    driver: bridge
```

- **Rationale:**

This configuration enables our Redpanda brokers and Redpanda Console to resolve each other's hostnames (e.g., redpanda-0, redpanda-1, redpanda-2) within the Docker network while keeping internal traffic isolated from the host machine. By isolating the internal network, our system prevents exposure of internal traffic to external entities, thereby enhancing security.

4.4.1.2 Volumes

We configured three named volumes (redpanda-0, redpanda-1, and redpanda-2), each associated with the data directories of the three Redpanda broker containers. The null value indicates that these volumes are defined without specifying a custom driver or external configuration, defaulting to Docker's local volume driver.

Volume configurations are a crucial component of our Docker-Compose setup, providing persistent storage to ensure service continuity and data integrity during container restarts or interruptions. By persisting data for each broker's directory (/var/lib/redpanda/data), this configuration protects critical data, including Kafka topics, metadata, and logs, ensuring reliable and uninterrupted system operations.

- **Configuration:**

```
yaml
volumes:
  redpanda-0: null
  redpanda-1: null
  redpanda-2: null
```

- **Rationale:**

Persistent storage is critical for maintaining data integrity and durability across container restarts, particularly for high-availability systems that manage crucial data like Redpanda topics and logs. This configuration is vital for maintaining data durability in our IOT systems.

4.4.1.3 Shared Command Configurations

Shared Command Configurations in our Docker Compose setup refer to command-line arguments applied to multiple containers to maintain consistent behavior across the system. These commands, defined in the command section of the Compose file, specify how services operate during runtime.

The shared commands include:

- **--mode dev-container:** Configures the container for development by applying default settings optimized for lower memory and resource usage.
- **--smp 1:** Restricts the broker to utilize only a single CPU core, ensuring efficient resource allocation.
- **--default-log-level=info:** Sets the logging level to provide meaningful debugging information without generating excessive verbosity.
- **Rationale:**

Shared Command Configurations in the Docker Compose setup lies in ensuring consistent, efficient, and manageable runtime behavior across all containers. By using `--mode dev-container`, the setup optimizes the containers for development environments, where reduced memory and resource usage are often necessary. This mode simplifies configuration by applying default settings that are lightweight and suitable for testing purposes, without compromising functionality.

The inclusion of `--smp 1` further enhances resource efficiency by limiting each broker to a single CPU core. This is particularly beneficial in constrained environments such as local machines or CI pipelines, where balancing performance with resource availability is critical.

Finally, the `--default-log-level=info` command ensures that the system produces actionable logging data without overwhelming developers with excessive detail. This balance is critical in a client-server setup, as clear and concise logs enable faster debugging and monitoring of client-server interactions, ensuring smooth operations and prompt issue resolution. Collectively, these configurations provide a foundation for a robust and efficient client-server architecture, capable of scaling while maintaining performance and reliability.\

4.4.2 Broker-Specific Configurations

Broker-specific configurations define the operational behavior of individual brokers, ensuring proper cluster formation and facilitating both internal and external interactions. These configurations are critical for scalability, availability, and the overall reliability of the Redpanda cluster.

This configuration enhances networking flexibility and security. It allows services within the same network to connect efficiently while providing external clients a reliable point of entry. This setup is essential for scalability, as it supports robust communication patterns across diverse client-server IOT environments.

- **Configuration:**

```
yaml
redpanda-0:
  command:
    - redpanda
    - start
    - --kafka-addr internal://0.0.0.0:9092,external://0.0.0.0:19092
    - --advertise-kafka-addr internal://redpanda-
      0:9092,external://${PUBLIC_IP}:19092
    ...
```

The configuration for redpanda-0 defines how the first Redpanda broker operates and communicates within the system. The command section specifies the startup instructions for the broker. The `--kafka-addr` argument configures the broker to listen for Kafka API requests on two separate address types: `internal://0.0.0.0:9092` for internal traffic within the Docker network and `external://0.0.0.0:19092` for traffic originating outside the network. This dual addressing ensures that the broker can handle both internal service communications and external client requests.

The `--advertise-kafka-addr` argument determines the addresses that the broker advertises to clients, enabling proper routing of requests. For internal clients within the Docker network, the broker advertises `internal://redpanda-0:9092`, using its hostname to facilitate DNS-based communication. For external clients, the broker advertises `external://${PUBLIC_IP}:19092`, where `${PUBLIC_IP}` is dynamically replaced with the host machine's public IP. This setup

ensures that clients, regardless of their origin, can connect seamlessly to the broker without manual reconfiguration.

- **Rationale:**

The rationale for using this setup in a client-server model, particularly in an IoT context, lies in its ability to efficiently handle diverse communication requirements and ensure seamless connectivity between devices and the central server. The configuration leverages dual addressing (`--kafka-addr` and `--advertise-kafka-addr`) to support both internal and external communication. In an IoT ecosystem, internal addresses (`internal://0.0.0.0:9092`) facilitate streamlined communication among services within the local Docker network, such as brokers and message processors. Meanwhile, external addresses (`external://${PUBLIC_IP}:19092`) enable IoT devices and remote clients to securely connect to the server, ensuring uninterrupted data flow from sensors and edge devices to the central broker.

This separation of internal and external traffic ensures optimized resource utilization and reduces the risk of network conflicts or bottlenecks. IoT environments often involve a vast number of devices sending data to the server, and the ability to advertise specific addresses to internal and external clients ensures that the broker remains accessible and adaptable to network topology changes. Moreover, the use of dynamic public IPs for external communication (`${PUBLIC_IP}`) provides flexibility, allowing devices to connect without requiring manual reconfiguration when the server's public IP changes.

4.4.3 Networking Considerations

Networking configurations ensure proper communication between services within the Docker network and external clients. They separate internal and external traffic to enhance efficiency and security.

4.4.3.1 Dual Kafka API Addressing

This setup optimizes resource utilization, enhances security, and supports scalability. It allows the our brokers to handle high volumes of data from sensor nodes while maintaining robust internal communication between backend services, making it an ideal configuration for real-time IoT applications and distributed client-server architectures.

- **Configuration:**

```
yaml
```

```
--kafka-addr internal://0.0.0.0:9092,external://0.0.0.0:19092
```

```
--advertise-kafka-addr internal://redpanda-0:9092,external://${PUBLIC_IP}:19092
```

The `--kafka-addr` configuration defines two distinct interfaces for the broker to listen on: an internal address (`internal://0.0.0.0:9092`) for communications within the Docker network and an external address (`external://0.0.0.0:19092`) for clients and devices outside the network. This separation is vital in IoT ecosystems, where numerous devices and applications rely on a central server to process and route data.

The `--advertise-kafka-addr` argument further enhances connectivity by defining how the broker advertises itself to clients. For internal services, the broker advertises `internal://redpanda-0:9092`, allowing efficient DNS-based communication within the Docker environment. For external IoT devices and applications, it advertises `external://${PUBLIC_IP}:19092`, ensuring that remote clients can dynamically locate and connect to the broker using the server's public IP address.

- **Rationale:**

Using this setup in a client-server model, lies in its ability to manage diverse communication needs while maintaining scalability, security, and efficiency. The configuration of `--kafka-addr` and `--advertise-kafka-addr` separates internal and external traffic, allowing seamless communication within the local network and reliable connectivity for external sensor nodes. In an IoT context, where sensors, edge devices, and clients continuously send data to a central server, the internal address ensures optimized communication between services like brokers and processors within the Docker network, enhancing system performance. Meanwhile, the external address provides a stable endpoint for remote IoT devices and clients, dynamically adapting to changes in the server's public IP. This dual-addressing approach ensures that the system is resilient and adaptable to the high traffic and dynamic nature of IoT networks. By advertising distinct addresses for internal and

external connections, the setup prevents conflicts, improves resource allocation, and facilitates secure, scalable operations essential for real-time data exchange in our system.

4.4.3.2 HTTP Proxy (Pandaproxy)

This setup simplifies integration for IoT devices and external applications by providing a user-friendly HTTP interface. It enables seamless data ingestion and processing, supports scalability, and ensures secure communication, making it an essential component of modern IoT and our client-server system.

- **Configuration:**

```
yaml
--pandaproxy-addr internal://0.0.0.0:8082,external://0.0.0.0:18082
--advertise-pandaproxy-addr internal://redpanda-
0:8082,external://${PUBLIC_IP}:18082
```

The `--pandaproxy-addr` configuration defines the endpoints where the Pandaproxy service listens for HTTP traffic. The internal address (`internal://0.0.0.0:8082`) facilitates communication between services within the Docker network, such as backend applications or analytics services, while the external address (`external://0.0.0.0:18082`) is used by external clients and IoT devices to connect to the server. This separation of traffic ensures that internal and external communications remain efficient and secure.

The `--advertise-pandaproxy-addr` argument specifies how the Pandaproxy service advertises itself to clients. Internally, it uses `internal://redpanda-0:8082`, allowing services within the Docker network to resolve and connect to the proxy. Externally, it advertises `external://${PUBLIC_IP}:18082`, enabling IoT devices and remote clients to connect using the server's public IP.

- **Rationale:**

Using this setup in a client-server model and IoT systems lies in its ability to simplify and optimize communication between lightweight sensor nodes, and the central server. The -

-pandaproxy-addr and --advertise-pandaproxy-addr configurations provide a dual-interface setup: the internal address ensures efficient communication among backend services within the Docker network, while the external address offers a reliable and accessible endpoint for IoT devices and external applications via the HTTP protocol. In IoT ecosystems, where devices often have limited computational resources and depend on RESTful APIs for data exchange, this setup eliminates the complexity of Kafka clients, making integration straightforward and efficient.

4.4.4 Console Configuration

The console configuration integrates a web-based interface for monitoring and managing the Redpanda cluster. It simplifies management while enhancing security and documentation.

4.4.4.1 Docker image

The Docker image configuration ensures that the Redpanda Console is deployed with consistent behavior and easy maintenance. We use the Redpanda Console image version (v2.7.2) which is the newest one.

- **Configuration:**

```
yaml
image: docker.redpanda.com/redpandadata/console:v2.7.2
```

4.4.4.2 Environment Variables:

The environment variables are crucial for configuring the Redpanda Console, enabling secure and efficient interactions with the cluster. Each setting plays a specific role in defining how the console connects to brokers, authenticates users, and provides access for monitoring and management.

- **Configuration:**

- a. Kafka Brokers:**

This setup configures environment variables to define how the Redpanda Console connects to the Redpanda brokers in a our client-server system.

```
yaml
```

```
environment: CONFIG_FILEPATH: ${CONFIG_FILEPATH:-/tmp/config.yml}  
CONSOLE_CONFIG_FILE: | kafka: brokers: ["redpanda-0:9092"]
```

The `CONFIG_FILEPATH` variable specifies the path to the configuration file used by the Redpanda Console. The default value is set to `/tmp/config.yml`, ensuring the console has a fallback location if no custom file path is provided. The `CONSOLE_CONFIG_FILE` variable embeds the configuration directly, defining the Kafka broker(s) the console will connect to. In this case, the console is set to connect to `redpanda-0:9092`, the internal address of the first broker in the cluster.

In our client-server system, this setup ensures that the console can seamlessly interact with the Redpanda cluster, facilitating centralized monitoring and management of data streams. In an IoT context, where large volumes of data from sensors and devices are ingested and processed, this configuration allows administrators to monitor broker performance, track topic activity, and manage system health efficiently.

b. SASL Authentication:

This SASL (Simple Authentication and Security Layer) configuration enables secure authentication between clients and the server in the client-server model.

```
yaml  
  
sasl:  
  
  enabled: true  
  
  username: superuser  
  
  password: secretpassword  
  
  mechanism: SCRAM-SHA-256
```

By setting `enabled: true`, it ensures that all connections to the server require authentication. The credentials (`username: superuser` and `password: secretpassword`) define an authorized user, and the authentication mechanism (`SCRAM-SHA-256`) specifies the secure hashing algorithm used to validate these credentials.

In the client-server model, this setup plays a critical role in ensuring that only authorized clients can interact with the Kafka brokers. It protects sensitive operations, such as data ingestion and retrieval, from unauthorized access, thus maintaining the integrity and confidentiality of the system. Additionally, the use of `SCRAM-SHA-256`, a robust and industry-standard hashing algorithm, provides a strong layer of security against potential threats like brute force or replay attacks.

c. Ports

This setting determines the port 8080 on which the Redpanda Console is accessible. It facilitates web-based access for developers and operators to manage and monitor the Redpanda cluster. Allowing developers to access the web interface via `http://localhost:8080` for cluster monitoring and management.

```
yaml  
  
ports:  
  
  - 8080:8080
```

4.4.5 Summary

The Redpanda Docker-Compose configuration is designed to provide a modular, secure, and scalable environment suitable for development and testing. By clearly delineating responsibilities across global configurations, broker-specific settings, networking considerations, and console management, the setup ensures a balance of simplicity, observability, and performance. These configurations collectively support a robust data streaming system that can be seamlessly adapted for production use.

Chapter 5

User Manual

This chapter provides a step-by-step guide for deploying, running, and managing the IoT Environmental Monitoring System utilizing Redpanda Data Streaming and the GAMPS Compression Algorithm. Follow the instructions below to set up and operate the system effectively.

5.1 Prerequisites

Before deploying the system, ensure that the following tools and dependencies are installed:

- Python 3.x
- Docker and Docker Compose
- Access to the dataset (see installation steps)

5.2 Installation

5.2.1 Clone the Repository

Download the project repository and navigate to the project folder:

```
1 git clone  
   https://github.com/tuanle186/redpanda_datastreaming_and_compression.git  
2 cd redpanda_datastreaming_and_compression
```

5.2.2 Install Python Dependencies

```
1 pip install -r requirements.txt
```

5.2.3 Add the Dataset

Download the dataset from [this link](#) and place it in the correct directory as described in the following file structure.

```

.
├── .gitignore
├── bootstrap.yml
├── config/
├── data/
│   ├── processed/
│   │   ├── data.csv
│   │   ├── sorted/
│   │   │   ├── moteid_1.csv
│   │   │   ├── moteid_2.csv
│   │   │   └── ...
│   │   └── unsorted/
│   │       └── ...
│   └── raw/
│       └── data.txt
├── docker-compose.yml
└── src/
    ├── client_producer/
    │   └── client.py
    ├── main.py
    └── server_consumer/
        ├── data/
        │   ├── compressed/
        │   ├── decompressed/
        │   └── raw_data.txt
        ├── MultiSensorDataGrouper.py
        └── server.py

```

5.3 Running the System

5.3.1 Start Redpanda

Use the following command to start the Redpanda instance via Docker Compose:

```
1 python src/main.py redpanda
```

5.3.2 Run the Client

Open a new terminal and run the client script to produce data to the Redpanda broker:

```
1 python src/main.py client --redpanda_ip_address <REDPANDA_IP_ADDRESS>
```

Replace `<REDPANDA_IP_ADDRESS>` with the IP address of your Redpanda instance (default: localhost).

5.3.3 Run the Server

In another terminal window, execute the following command to consume and process data from the Redpanda broker:

```
1 python src/main.py server --redpanda_ip_address <REDPANDA_IP_ADDRESS>
```

Ensure the `<REDPANDA_IP_ADDRESS>` matches the client's.

5.3.4 (Optional) Server Data Compression

To compress data manually, use:

```
1 python src/main.py server_data_compression
```

5.3.5 (Optional) Server Data Compression

To decompress data:

```
1 python src/main.py server_data_decompression
```

5.3.6 Stop Redpanda

Once you are finished, stop the Redpanda instance by running:

```
1 python src/main.py stop_redpanda
```


Chapter 6

Project Evaluation and Demonstration

6.1 Redpanda Data Streaming

6.1.1 User Manual Demonstration

6.1.1.1 Introduction

- **Objective:** Demonstrate how to set up and operate the system using Redpanda for message streaming and a client-server application.
- **Prerequisites:**
 - Docker and Docker Compose installed.
 - Python environment with necessary dependencies.
 - Source code repository cloned locally.

6.1.1.2 Setup Environment

Verify Docker installation:

```
1 docker --version
2 docker compose --version
```

Confirm Python version and install dependencies:

```
1 python --version
2 pip install -r requirements.txt
```

6.1.1.3 Starting Redpanda Instance

Host running Redpanda Ipconfig:

```
Ethernet adapter Radmin VPN:

Connection-specific DNS Suffix . : 
IPv6 Address. . . . . : fdfd::1a47:7865
Link-local IPv6 Address . . . . : fe80::65a8:c6c1:99ac:6378%4
IPv4 Address. . . . . : 26.71.120.101
Subnet Mask . . . . . : 255.0.0.0
Default Gateway . . . . . : 26.0.0.1
```

Figure 6.7 – Host running Redpanda Ipconfig

Build and Start Redpanda by running this command:

```
1 python src/main.py redpanda
```

If the command output is similar to the following, it indicates that the Redpanda instance has been successfully built and is ready to run with the IP address 192.168.1.11

```
D:\tai_lieu_dai_hoc\HK9\Đồ án đã ngành\Final_V1\redpanda_datastreaming_and_compresison>python src/main.py redpanda
Running in REDPANDA mode...
Starting Redpanda using Docker Compose...
[+] Running 4/4
✔Container redpanda-0          Started 2.7s
✔Container redpanda-2          Started 2.9s
✔Container redpanda-1          Started 2.8s
✔Container redpanda-console    Started 2.5s
```

Figure 6.8 – Docker Compose successfully and all Containers are running

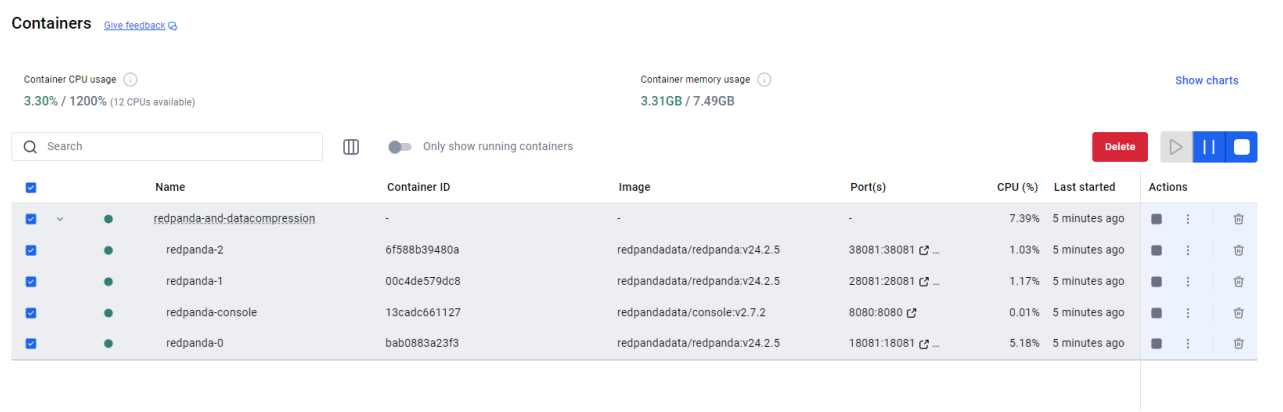


Figure 6.9 – Showing container’s ports and CPU usage

6.1.1.4 Running the Client

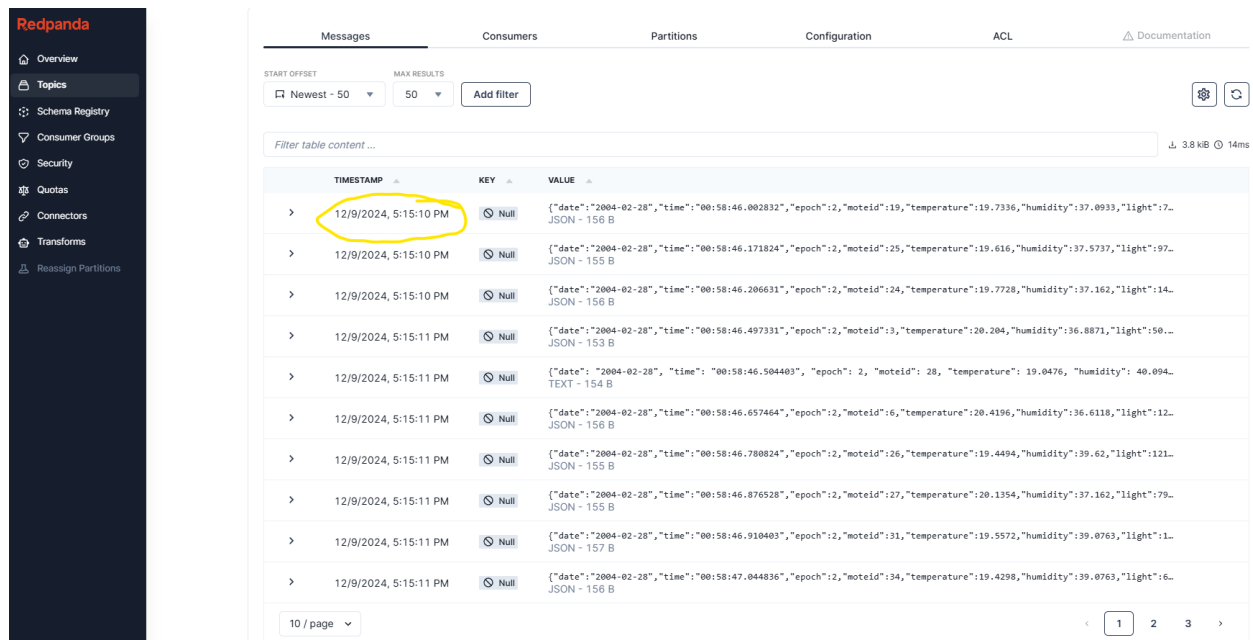
Run the client mode using the following command. The client will send data from all sensor nodes to Redpanda every 31 seconds:

1 python src/main.py client --redpanda_ip_address 192.168.1.11

```
Running in CLIENT mode...
2024-12-09 17:15:10,513 - INFO - Data loaded successfully.
2024-12-09 17:15:10,519 - INFO - Connected to Redpanda broker: 192.168.1.11:19092
2024-12-09 17:15:10,536 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.002832', 'epoch': 2, 'moteid': 19, 'temperature': 19.7336, 'humidity': 37.0933, 'light': 71.76, 'voltage': 2.69964}
2024-12-09 17:15:10,711 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.171824', 'epoch': 2, 'moteid': 25, 'temperature': 19.616, 'humidity': 37.5737, 'light': 97.52, 'voltage': 2.69964}
2024-12-09 17:15:10,748 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.206631', 'epoch': 2, 'moteid': 24, 'temperature': 19.7728, 'humidity': 37.162, 'light': 143.52, 'voltage': 2.71196}
2024-12-09 17:15:11,041 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.497331', 'epoch': 2, 'moteid': 3, 'temperature': 20.204, 'humidity': 36.8871, 'light': 50.6, 'voltage': 2.69964}
2024-12-09 17:15:11,050 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.504403', 'epoch': 2, 'moteid': 28, 'temperature': 19.0476, 'humidity': 40.0945, 'light': nan, 'voltage': 2.80151}
2024-12-09 17:15:11,204 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.657464', 'epoch': 2, 'moteid': 6, 'temperature': 20.4196, 'humidity': 36.6118, 'light': 121.44, 'voltage': 2.65143}
2024-12-09 17:15:11,329 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.780824', 'epoch': 2, 'moteid': 26, 'temperature': 19.4494, 'humidity': 39.62, 'light': 121.44, 'voltage': 2.69964}
2024-12-09 17:15:11,426 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.876528', 'epoch': 2, 'moteid': 27, 'temperature': 20.1354, 'humidity': 37.162, 'light': 79.12, 'voltage': 2.71196}
2024-12-09 17:15:11,463 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:46.910403', 'epoch': 2, 'moteid': 31, 'temperature': 19.5572, 'humidity': 39.0763, 'light': 150.88, 'voltage': 2.69964}
2024-12-09 17:15:11,599 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:47.044836', 'epoch': 2, 'moteid': 34, 'temperature': 19.4298, 'humidity': 39.0763, 'light': 60.72, 'voltage': 2.68742}
2024-12-09 17:15:11,688 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:47.132121', 'epoch': 2, 'moteid': 43, 'temperature': 19.7532, 'humidity': 35.853, 'light': 478.4, 'voltage': 2.65143}
2024-12-09 17:15:11,737 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:47.179192', 'epoch': 2, 'moteid': 20, 'temperature': 21.576, 'humidity': 33.4905, 'light': 128.8, 'voltage': 3.15915}
2024-12-09 17:15:11,783 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:58:47.222833', 'epoch': 2, 'moteid': 40, 'temperature': 19.5964, 'humidity': 37.5737, 'light': 172.96, 'voltage': 2.68742}
```

Figure 6.10 – Client sends data from sensor nodes to Redpanda

The blue circle indicates that the message was produced on 12-09-2024 at 17:15:10,536.



Messages	Consumers	Partitions	Configuration	ACL	Documentation
START OFFSET: Newest - 50 MAX RESULTS: 50 Add filter					
Filter table content ... 3.8 kB 14ms					
TIMESTAMP	KEY	VALUE			
12/9/2024, 5:15:10 PM	Null	["date": "2004-02-28", "time": "00:58:46.002832", "epoch": 2, "moteid": 19, "temperature": 19.7336, "humidity": 37.0933, "light": 71.76, "voltage": 2.69964]			
12/9/2024, 5:15:10 PM	Null	["date": "2004-02-28", "time": "00:58:46.171824", "epoch": 2, "moteid": 25, "temperature": 19.616, "humidity": 37.5737, "light": 97.52, "voltage": 2.69964]			
12/9/2024, 5:15:10 PM	Null	["date": "2004-02-28", "time": "00:58:46.206631", "epoch": 2, "moteid": 24, "temperature": 19.7728, "humidity": 37.162, "light": 143.52, "voltage": 2.71196]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.497331", "epoch": 2, "moteid": 3, "temperature": 20.204, "humidity": 36.8871, "light": 50.6, "voltage": 2.69964]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.504403", "epoch": 2, "moteid": 28, "temperature": 19.0476, "humidity": 40.0945, "light": null, "voltage": 2.80151]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.657464", "epoch": 2, "moteid": 6, "temperature": 20.4196, "humidity": 36.6118, "light": 121.44, "voltage": 2.65143]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.780824", "epoch": 2, "moteid": 26, "temperature": 19.4494, "humidity": 39.62, "light": 121.44, "voltage": 2.69964]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.876528", "epoch": 2, "moteid": 27, "temperature": 20.1354, "humidity": 37.162, "light": 79.12, "voltage": 2.71196]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:46.910403", "epoch": 2, "moteid": 31, "temperature": 19.5572, "humidity": 39.0763, "light": 150.88, "voltage": 2.69964]			
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:47.044836", "epoch": 2, "moteid": 34, "temperature": 19.4298, "humidity": 39.0763, "light": 60.72, "voltage": 2.68742]			

Figure 6.11 – Redpanda instantly gets data

The yellow circle indicates that the message was sent to Redpanda on 12-09-2024 at 17:15:10.

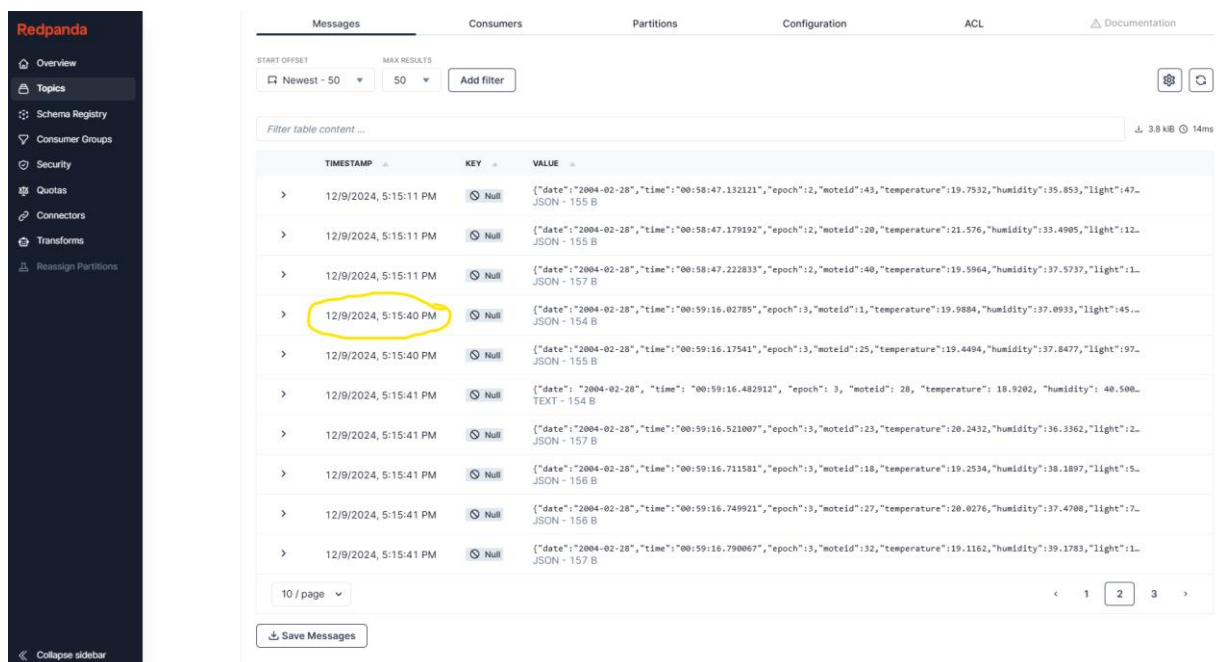
```

2024-12-09 17:15:40,784 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,094 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.482912', 'epoch': 3, 'moteid': 28, 'temp
erature': 18.9202, 'humidity': 40.5004, 'light': nan, 'voltage': 2.80151}
2024-12-09 17:15:41,094 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,135 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.521007', 'epoch': 3, 'moteid': 23, 'temp
erature': 20.2432, 'humidity': 36.3362, 'light': 235.52, 'voltage': 2.69964}
2024-12-09 17:15:41,135 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,327 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.711581', 'epoch': 3, 'moteid': 18, 'temp
erature': 19.2534, 'humidity': 38.1897, 'light': 57.04, 'voltage': 2.50599}
2024-12-09 17:15:41,327 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,368 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.749921', 'epoch': 3, 'moteid': 27, 'temp
erature': 20.0276, 'humidity': 37.4708, 'light': 79.12, 'voltage': 2.69964}
2024-12-09 17:15:41,410 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.790067', 'epoch': 3, 'moteid': 32, 'temp
erature': 19.1162, 'humidity': 39.1783, 'light': 114.08, 'voltage': 2.69964}
2024-12-09 17:15:41,410 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,410 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,476 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.853347', 'epoch': 3, 'moteid': 22, 'temp
erature': 19.6356, 'humidity': 37.0246, 'light': 82.8, 'voltage': 2.65143}
2024-12-09 17:15:41,476 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,559 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.933957', 'epoch': 3, 'moteid': 24, 'temp
erature': 19.6356, 'humidity': 37.3336, 'light': 143.52, 'voltage': 2.71196}
2024-12-09 17:15:41,559 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,612 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:16.984189', 'epoch': 3, 'moteid': 26, 'temp
erature': 19.3318, 'humidity': 39.9929, 'light': 121.44, 'voltage': 2.68742}
2024-12-09 17:15:41,612 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,734 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:17.103111', 'epoch': 3, 'moteid': 51, 'temp
erature': 18.6066, 'humidity': 37.7792, 'light': 143.52, 'voltage': 2.68742}
2024-12-09 17:15:41,734 - INFO - Message delivered to sensor-data [0]
2024-12-09 17:15:41,775 - INFO - Producing message: {'date': '2004-02-28', 'time': '00:59:17.141303', 'epoch': 3, 'moteid': 52, 'temp
erature': 18.3028, 'humidity': 38.7698, 'light': 64.4, 'voltage': 2.65143}

```

Figure 6.12 – Client sends data from sensor nodes to Redpanda after next 31 seconds

The blue circle indicates that the message was produced in next 31 seconds on 12-09-2024 at 17:15:40,784.



TIMESTAMP	KEY	VALUE
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:47.132121", "epoch": 2, "moteid": 43, "temperature": 19.7532, "humidity": 35.853, "light": 47....
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:47.179192", "epoch": 2, "moteid": 20, "temperature": 21.576, "humidity": 33.4905, "light": 12....
12/9/2024, 5:15:11 PM	Null	["date": "2004-02-28", "time": "00:58:47.222833", "epoch": 2, "moteid": 40, "temperature": 19.5964, "humidity": 37.5737, "light": 1....
12/9/2024, 5:15:40 PM	Null	["date": "2004-02-28", "time": "00:59:16.02785", "epoch": 3, "moteid": 1, "temperature": 19.9884, "humidity": 37.0933, "light": 45....
12/9/2024, 5:15:40 PM	Null	["date": "2004-02-28", "time": "00:59:16.17541", "epoch": 3, "moteid": 25, "temperature": 19.4494, "humidity": 37.8477, "light": 97....
12/9/2024, 5:15:41 PM	Null	["date": "2004-02-28", "time": "00:59:16.482912", "epoch": 3, "moteid": 28, "temperature": 18.9202, "humidity": 40.5004, "light": 40.5004. TEXT - 154 B
12/9/2024, 5:15:41 PM	Null	["date": "2004-02-28", "time": "00:59:16.521007", "epoch": 3, "moteid": 23, "temperature": 20.2432, "humidity": 36.3362, "light": 2....
12/9/2024, 5:15:41 PM	Null	["date": "2004-02-28", "time": "00:59:16.711581", "epoch": 3, "moteid": 18, "temperature": 19.2534, "humidity": 38.1897, "light": 5....
12/9/2024, 5:15:41 PM	Null	["date": "2004-02-28", "time": "00:59:16.749921", "epoch": 3, "moteid": 27, "temperature": 20.0276, "humidity": 37.4708, "light": 7....
12/9/2024, 5:15:41 PM	Null	["date": "2004-02-28", "time": "00:59:16.790067", "epoch": 3, "moteid": 32, "temperature": 19.1162, "humidity": 39.1783, "light": 1....

Figure 6.13 – Redpanda instantly gets data

The yellow circle indicates that the message was sent to Redpanda on 12-09-2024 at 17:15:40.

6.1.1.5 Running the Server

Run the server mode using the following command. The server will immediately start listening and consuming data from Redpanda if there is any unconsumed data. After 5 hours, it will automatically perform compression:

```
1 python src/main.py server --redpanda_ip_address 192.168.1.11
```

The two pictures below demonstrate that the data is consumed by the server almost instantly.

```
2024-12-09 17:15:21,563 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.002832', 'epoch': 2, 'metoid': 19, 'temperature': 19.7336, 'humidity': 37.0933, 'light': 71.76, 'voltage': 2.69964}
2024-12-09 17:15:21,588 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.002832', 'epoch': 2, 'metoid': 19, 'temperature': 19.7336, 'humidity': 37.0933, 'light': 71.76, 'voltage': 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.171824", "epoch": 2, "metoid": 25, "temperature": 19.616, "humidity": 37.5737, "light": 97.52, "voltage": 2.69964}'
2024-12-09 17:15:21,589 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.171824', 'epoch': 2, 'metoid': 25, 'temperature': 19.616, 'humidity': 37.5737, 'light': 97.52, 'voltage': 2.69964}
2024-12-09 17:15:21,601 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.171824', 'epoch': 2, 'metoid': 25, 'temperature': 19.616, 'humidity': 37.5737, 'light': 97.52, 'voltage': 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.206631", "epoch": 2, "metoid": 24, "temperature": 19.7728, "humidity": 37.162, "light": 143.52, "voltage": 2.71196}'
2024-12-09 17:15:21,602 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.206631', 'epoch': 2, 'metoid': 24, 'temperature': 19.7728, 'humidity': 37.162, "light": 143.52, "voltage": 2.71196}
2024-12-09 17:15:21,613 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.206631', 'epoch': 2, 'metoid': 24, 'temperature': 19.7728, 'humidity': 37.162, "light": 143.52, "voltage": 2.71196}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.497331", "epoch": 2, "metoid": 3, "temperature": 20.204, "humidity": 36.8871, "light": 50.6, "voltage": 2.69964}'
2024-12-09 17:15:21,613 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.497331', 'epoch': 2, 'metoid': 3, 'temperature': 20.204, "humidity": 36.8871, "light": 50.6, "voltage": 2.69964}
2024-12-09 17:15:21,627 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.497331', 'epoch': 2, 'metoid': 3, 'temperature': 20.204, "humidity": 36.8871, "light": 50.6, "voltage": 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.594083", "epoch": 2, "metoid": 28, "temperature": 19.0476, "humidity": 40.0945, "light": nan, "voltage": 2.80151}'
2024-12-09 17:15:21,627 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.594083', 'epoch': 2, 'metoid': 28, "temperature": 19.0476, "humidity": 40.0945, "light": nan, "voltage": 2.80151}
2024-12-09 17:15:21,638 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.594083', 'epoch': 2, "metoid": 28, "temperature": 19.0476, "humidity": 40.0945, "light": nan, "voltage": 2.80151}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.657464", "epoch": 2, "metoid": 6, "temperature": 20.4196, "humidity": 36.6118, "light": 121.44, "voltage": 2.65143}'
2024-12-09 17:15:21,638 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.657464', 'epoch': 2, "metoid": 6, "temperature": 20.4196, "humidity": 36.6118, "light": 121.44, "voltage": 2.65143}
2024-12-09 17:15:21,651 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.657464', 'epoch': 2, "metoid": 6, "temperature": 20.4196, "humidity": 36.6118, "light": 121.44, "voltage": 2.65143}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.780824", "epoch": 2, "metoid": 26, "temperature": 19.4494, "humidity": 39.62, "light": 121.44, "voltage": 2.69964}'
2024-12-09 17:15:21,651 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.780824', 'epoch': 2, "metoid": 26, "temperature": 19.4494, "humidity": 39.62, "light": 121.44, "voltage": 2.69964}
2024-12-09 17:15:21,663 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.780824', 'epoch': 2, "metoid": 26, "temperature": 19.4494, "humidity": 39.62, "light": 121.44, "voltage": 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.876528", "epoch": 2, "metoid": 27, "temperature": 20.1354, "humidity": 37.162, "light": 79.12, "voltage": 2.71196}'
2024-12-09 17:15:21,663 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.876528', 'epoch': 2, "metoid": 27, "temperature": 20.1354, "humidity": 37.162, "light": 79.12, "voltage": 2.71196}
2024-12-09 17:15:21,676 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.876528', 'epoch': 2, "metoid": 27, "temperature": 20.1354, "humidity": 37.162, "light": 79.12, "voltage": 2.71196}
Received message: b'{"date": "2004-02-28", "time": "00:58:46.910403", "epoch": 2, "metoid": 31, "temperature": 19.5572, "humidity": 39.0763, "light": 150.88, "voltage": 2.69964}'
2024-12-09 17:15:21,676 - INFO - Received data: {'date': '2004-02-28', 'time': '00:58:46.910403', 'epoch': 2, "metoid": 31, "temperature": 19.5572, "humidity": 39.0763, "light": 150.88, "voltage": 2.69964}
2024-12-09 17:15:21,689 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:58:46.910403', 'epoch': 2, "metoid": 31, "temperature": 19.5572, "humidity": 39.0763, "light": 150.88, "voltage": 2.69964}
```

Figure 6.14 –Server instantly consumes data

The yellow circle indicates that the message was consume from Redpanda by server on 12-09-2024 at 17:15:21,563 seconds.

```
2024-12-09 17:15:47,185 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.02785', 'epoch': 3, 'metoid': 1, 'temperature': 19.9884, 'humidity': 37.0933, 'light': 45.08, 'voltage': 2.69964}
2024-12-09 17:15:47,226 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.02785', 'epoch': 3, 'metoid': 1, 'temperature': 19.9884, 'humidity': 37.0933, 'light': 45.08, 'voltage': 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.17541", "epoch": 3, "metoid": 25, "temperature": 19.4494, "humidity": 37.8477, "light": 97.52, "voltage": 2.69964}'
2024-12-09 17:15:47,335 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.17541', 'epoch': 3, "metoid": 25, "temperature": 19.4494, "humidity": 37.8477, "light": 97.52, "voltage": 2.69964}
2024-12-09 17:15:47,346 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.17541', 'epoch': 3, "metoid": 25, "temperature": 19.4494, "humidity": 37.8477, "light": 97.52, "voltage": 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.482912", "epoch": 3, "metoid": 28, "temperature": 18.9202, "humidity": 40.5004, "light": nan, "voltage": 2.80151}'
2024-12-09 17:15:47,642 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.482912', 'epoch': 3, "metoid": 28, "temperature": 18.9202, "humidity": 40.5004, "light": nan, "voltage": 2.80151}
2024-12-09 17:15:47,651 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.482912', 'epoch': 3, "metoid": 28, "temperature": 18.9202, "humidity": 40.5004, "light": nan, "voltage": 2.80151}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.521007", "epoch": 3, "metoid": 23, "temperature": 20.2432, "humidity": 36.3362, "light": 235.52, "voltage": 2.69964}'
2024-12-09 17:15:47,686 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.521007', 'epoch': 3, "metoid": 23, "temperature": 20.2432, "humidity": 36.3362, "light": 235.52, "voltage": 2.69964}
2024-12-09 17:15:47,697 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.521007', 'epoch': 3, "metoid": 23, "temperature": 20.2432, "humidity": 36.3362, "light": 235.52, "voltage": 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.711581", "epoch": 3, "metoid": 18, "temperature": 19.2534, "humidity": 38.1897, "light": 57.04, "voltage": 2.50599}'
2024-12-09 17:15:47,881 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.711581', 'epoch': 3, "metoid": 18, "temperature": 19.2534, "humidity": 38.1897, "light": 57.04, "voltage": 2.50599}
2024-12-09 17:15:47,893 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.711581', 'epoch': 3, "metoid": 18, "temperature": 19.2534, "humidity": 38.1897, "light": 57.04, "voltage": 2.50599}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.749921", "epoch": 3, "metoid": 27, "temperature": 20.0276, "humidity": 37.4788, "light": 79.12, "voltage": 2.69964}'
2024-12-09 17:15:47,920 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.749921', 'epoch': 3, "metoid": 27, "temperature": 20.0276, "humidity": 37.4788, "light": 79.12, "voltage": 2.69964}
2024-12-09 17:15:47,939 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.749921', 'epoch': 3, "metoid": 27, "temperature": 20.0276, "humidity": 37.4788, "light": 79.12, "voltage": 2.69964}
Received message: b'{"date": "2004-02-28", "time": "00:59:16.790867", "epoch": 3, "metoid": 32, "temperature": 19.1162, "humidity": 39.1783, "light": 114.08, "voltage": 2.69964}'
2024-12-09 17:15:47,954 - INFO - Received data: {'date': '2004-02-28', 'time': '00:59:16.790867', 'epoch': 3, "metoid": 32, "temperature": 19.1162, "humidity": 39.1783, "light": 114.08, "voltage": 2.69964}
2024-12-09 17:15:47,967 - ERROR - Message failed: {'date': '2004-02-28', 'time': '00:59:16.790867', 'epoch': 3, "metoid": 32, "temperature": 19.1162, "humidity": 39.1783, "light": 114.08, "voltage": 2.69964}
```

Figure 6.15 –Server instantly consumes data after next 31 seconds

The yellow circle indicates that the message was consume from Redpanda by server on 12-09-2024 at 17:15:47,185 seconds.

6.1.1.6 Running the Compression and Decompression

Run the manual compression mode using the following command. The server will perform compression immediately, bypassing the 5-hour waiting period while maintaining the same functionality.


```
1 python src/main.py server_data_compression
```

As shown in the picture below, the compression was successfully completed. Each attribute was compressed and stored in a separate text file corresponding to the attribute. The numbers underlined in yellow indicate the index of the compression.

```
PS D:\tai_lieu_dai_hoc\HK9\Đồ án đa ngành\Final_V1\redpanda_datastreaming_and_compresison> python src/main.py server_data_compression
Running in SERVER DATA COMPRESSION mode...
2024-12-09 17:09:53,226 - INFO - Attribute: temperature - Total memory cost after compression: 3012 buckets
2024-12-09 17:09:54,951 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_temperature_1.txt
2024-12-09 17:09:56,481 - INFO - Attribute: humidity - Total memory cost after compression: 3518 buckets
2024-12-09 17:09:58,180 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_humidity_1.txt
2024-12-09 17:09:59,975 - INFO - Attribute: light - Total memory cost after compression: 87827 buckets
2024-12-09 17:10:02,017 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_light_1.txt
2024-12-09 17:10:03,460 - INFO - Attribute: voltage - Total memory cost after compression: 59 buckets
2024-12-09 17:10:04,957 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_voltage_1.txt
```

Figure 6.16 –Server do compression first time

```
PS D:\tai_lieu_dai_hoc\HK9\Đồ án đa ngành\Final_V1\redpanda_datastreaming_and_compresison> python src/main.py server_data_compression
Running in SERVER DATA COMPRESSION mode...
2024-12-09 18:07:15,085 - INFO - Attribute: temperature - Total memory cost after compression: 3012 buckets
2024-12-09 18:07:16,660 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_temperature_2.txt
2024-12-09 18:07:18,316 - INFO - Attribute: humidity - Total memory cost after compression: 3518 buckets
2024-12-09 18:07:20,064 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_humidity_2.txt
2024-12-09 18:07:21,817 - INFO - Attribute: light - Total memory cost after compression: 87827 buckets
2024-12-09 18:07:24,030 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_light_2.txt
2024-12-09 18:07:25,444 - INFO - Attribute: voltage - Total memory cost after compression: 59 buckets
2024-12-09 18:07:26,868 - INFO - Compressed data written to ./src/server_consumer/data/compressed/compressed_voltage_2.txt
PS D:\tai_lieu_dai_hoc\HK9\Đồ án đa ngành\Final_V1\redpanda_datastreaming_and_compresison> []
```

Figure 6.17 –Server do compression second time

Run the manual decompression mode using the following command. The server will perform decompression immediately

```
1 python src/main.py server_data_decompression
```

We performed compression twice as described above, resulting in 2 blocks of compressed data. To run the decompression mode, you need to enter the number corresponding to the block of compressed data you want to decompress. As shown in the pictures below, I entered " 2" to perform the decompression on block number 2.

```
PS D:\tai_lieu_dai_hoc\HK9\Đồ án đa ngành\Final_V1\redpanda_datastreaming_and_compresison> python src/main.py server_data_decompression
Running in SERVER DATA DECOMPRESSION mode...
Enter the number of compressed data entries you want to decompress (1 to 2): 2
2024-12-09 18:09:40,331 - INFO - Decompressed data for attribute 'temperature' written to ./src/server_consumer/data/decompressed/decompressed_temperature_2.txt.
2024-12-09 18:09:41,940 - INFO - Decompressed data for attribute 'humidity' written to ./src/server_consumer/data/decompressed/decompressed_humidity_2.txt.
2024-12-09 18:09:43,772 - INFO - Decompressed data for attribute 'light' written to ./src/server_consumer/data/decompressed/decompressed_light_2.txt.
2024-12-09 18:09:45,230 - INFO - Decompressed data for attribute 'voltage' written to ./src/server_consumer/data/decompressed/decompressed_voltage_2.txt.
```

Figure 6.18 –Server do decompression

6.1.2 Requirement Evaluation

6.1.2.1 Evaluate Functional Requirements

As outlined in Section 6.1.1, we successfully deployed Redpanda and ran the client-server application. Data from the producer to Redpanda was transmitted almost instantaneously, with a very small latency of approximately **14ms**.

On the consumer (server) side, it took a few seconds to process and consume the data. As demonstrated in Section 6.1.1.5, during the first attempt, it took approximately **6 seconds** to consume the initial data batch. For subsequent data consumption, over a duration of **31 seconds**, the server took an average of **7 seconds** to consume a data size of **3.8KB**.

These results indicate that the system efficiently meets the functional requirements for streaming data and handling real-time environmental data from an IoT Gateway.

6.1.2.2 Evaluate Non-functional Requirements

Performance

Messages

Consumers

Partitions

Configuration

ACL

[Documentation](#)

START OFFSET

MAX RESULTS

Newest - 50

50

Add filter

Filter table content ...

7.62 KiB

10ms

	TIMESTAMP	KEY	VALUE
>	12/9/2024, 6:40:35 PM	Null	{"date": "2004-02-28", "time": "01:02:47.034673", "epoch": 10, "moteid": 19, "temperature": 19.0966, "humidity": 38.4287, "light": ... JSON - 157 B
>	12/9/2024, 6:41:04 PM	Null	{"date": "2004-02-28", "time": "01:03:16.024291", "epoch": 11, "moteid": 28, "temperature": 18.1852, "humidity": 41.98... TEXT - 155 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.133126", "epoch": 11, "moteid": 26, "temperature": 18.6164, "humidity": 41.478, "light": 1... JSON - 157 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.179903", "epoch": 11, "moteid": 36, "temperature": 18.9496, "humidity": 41.1414, "light": ... JSON - 157 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.33393", "epoch": 11, "moteid": 1, "temperature": 19.3024, "humidity": 38.4629, "light": 45... JSON - 155 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.399785", "epoch": 11, "moteid": 29, "temperature": 19.028, "humidity": 39.5521, "light": 1... JSON - 157 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.699203", "epoch": 11, "moteid": 19, "temperature": 19.0672, "humidity": 38.5652, "light": ... JSON - 157 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.886569", "epoch": 11, "moteid": 34, "temperature": 18.5576, "humidity": 41.1078, "light": ... JSON - 157 B
>	12/9/2024, 6:41:05 PM	Null	{"date": "2004-02-28", "time": "01:03:16.92293", "epoch": 11, "moteid": 21, "temperature": 19.371, "humidity": 37.6765, "light": 11... JSON - 156 B
>	12/9/2024, 6:41:06 PM	Null	{"date": "2004-02-28", "time": "01:03:16.981838", "epoch": 11, "moteid": 40, "temperature": 18.881, "humidity": 39.1443, "light": 1... JSON - 157 B

10 / page

<

1

2

3

4

5

>

Figure 6.19 –Data set to test performance

Our system, integrated with Redpanda, easily handles real-time data production and consumption, meeting the maximum data size of **5KB**. The latency for sending data to Redpanda is approximately **10ms**, ensuring near-instantaneous transmission.

On the consumer (server) side, it takes approximately **7 seconds** to process and consume the data. This performance remains efficient, as the test data used for evaluation was **7.6KB**, exceeding the defined maximum of **5KB** without any significant impact on system functionality. Meet this non-requirement

Low Latency

The system provides low-latency streaming, ensuring that 5KB data batches from the IoT Gateway are available in real time for the consumer. With a latency of approximately **10ms** for sending data to Redpanda, even with a data size of **7.6KB**, the system successfully meets this non-functional requirement.

Data Consistency

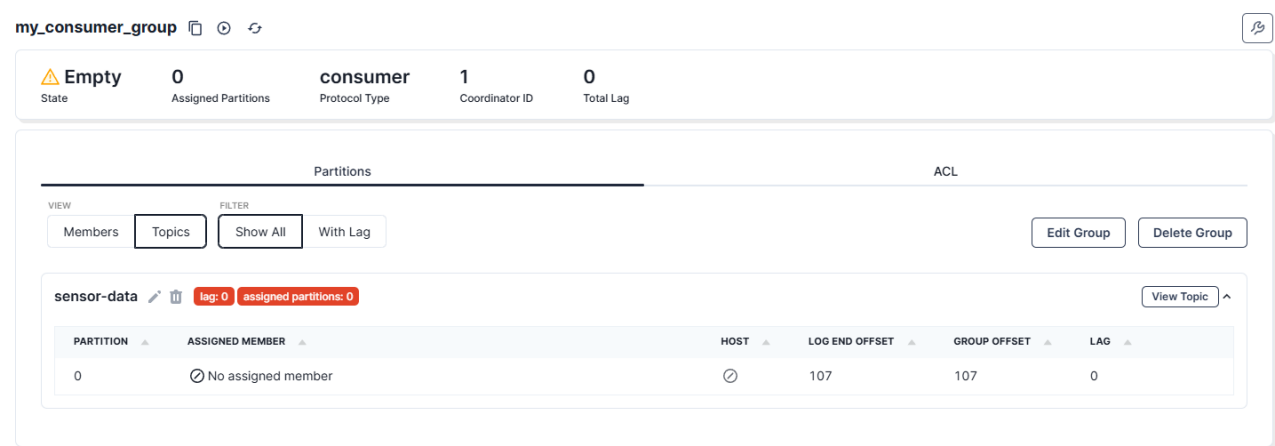


Figure 6.20 – LAG


```
src > server_consumer > data > raw_data.txt
```

70	2004-02-28	02:11:16.721059	147	1	18.8124	39.0082	43.24	2.69964
71	2004-02-28	02:12:46.278991	150	1	18.8124	39.0082	43.24	2.68742
72	2004-02-28	02:13:16.505098	151	1	18.8026	39.0082	43.24	2.68742
73	2004-02-28	02:14:46.685489	154	1	18.7928	38.9401	43.24	2.68742
74	2004-02-28	02:16:46.546677	158	1	18.7928	38.9401	43.24	2.68742
75	2004-02-28	02:18:46.231028	162	1	18.7732	39.0082	43.24	2.68742
76	2004-02-28	02:19:16.175456	163	1	18.7732	39.0082	43.24	2.68742
77	2004-02-28	02:19:46.991259	164	1	18.7928	38.872	43.24	2.69964
78	2004-02-28	02:20:46.870029	166	1	18.7732	38.9061	43.24	2.69964
79	2004-02-28	02:21:16.554543	167	1	18.7536	38.8379	43.24	2.68742
80	2004-02-28	02:21:46.303901	168	1	18.7732	38.8039	43.24	2.69964
81	2004-02-28	02:23:16.60315	171	1	18.7634	38.872	43.24	2.68742
82	2004-02-28	02:23:46.44942	172	1	18.7536	38.8039	43.24	2.69964
83	2004-02-28	02:24:46.059627	174	1	18.7242	38.8039	43.24	2.68742
84	2004-02-28	02:25:46.515577	176	1	18.734	38.8379	43.24	2.69964
85	2004-02-28	02:26:17.23131	177	1	18.7242	38.9401	43.24	2.69964
86	2004-02-28	02:30:16.107116	185	1	18.734	38.872	43.24	2.69964
87	2004-02-28	02:31:16.348393	187	1	18.7242	38.9401	43.24	2.68742
88	2004-02-28	02:31:46.107955	188	1	18.7144	38.9401	43.24	2.68742
89	2004-02-28	02:34:46.698204	194	1	18.6948	38.9401	43.24	2.68742
90	2004-02-28	02:35:46.720846	196	1	18.6948	38.9401	43.24	2.68742
91	2004-02-28	02:36:16.849224	197	1	18.685	38.872	43.24	2.69964
92	2004-02-28	02:37:46.514175	200	1	18.6654	38.872	43.24	2.68742
93	2004-02-28	02:40:16.927811	205	1	18.6556	38.872	43.24	2.68742
94	2004-02-28	02:40:46.12126	206	1	18.6458	38.872	43.24	2.68742
95	2004-02-28	02:41:16.692999	207	1	18.636	38.872	43.24	2.69964
96	2004-02-28	02:42:46.365373	210	1	18.636	38.8379	43.24	2.68742
97	2004-02-28	02:43:46.923849	212	1	18.636	38.872	43.24	2.68742
98	2004-02-28	02:46:46.166591	218	1	18.636	38.8039	43.24	2.68742
99	2004-02-28	02:47:46.1321	220	1	18.6164	38.872	43.24	2.69964
100	2004-02-28	02:50:16.186743	225	1	18.6066	38.9061	43.24	2.68742
101	2004-02-28	03:02:17.064743	249	1	18.5478	38.9401	43.24	2.67532
102	2004-02-28	03:05:46.58116	256	1	18.5184	38.9401	43.24	2.68742
103	2004-02-28	03:07:46.193901	260	1	18.5086	39.0082	43.24	2.68742
104	2004-02-28	03:08:16.336506	261	1	18.4988	39.0763	43.24	2.68742
105	2004-02-28	03:08:47.118704	262	1	18.4988	39.0082	43.24	2.69964
106	2004-02-28	03:10:17.006142	265	1	18.4988	39.0422	43.24	2.68742
107	2004-02-28	03:10:46.49789	266	1	18.5086	39.0422	43.24	2.68742

Figure 6.21 –Number of row consumed from Redpanda

The two pictures above demonstrate data consistency in the system.

Figure 3.20: The LAG value is 0, indicating that there is no difference between the last message produced by the producer and the last message consumed by the server. This confirms real-time synchronization between production and consumption.

Figure 3.21: The server has consumed **107 rows**, verifying that the data is consistent and there are no duplicates in the process.

So it meets the requirement that no data loss or duplication during transmission and storage

Fault Tolerance

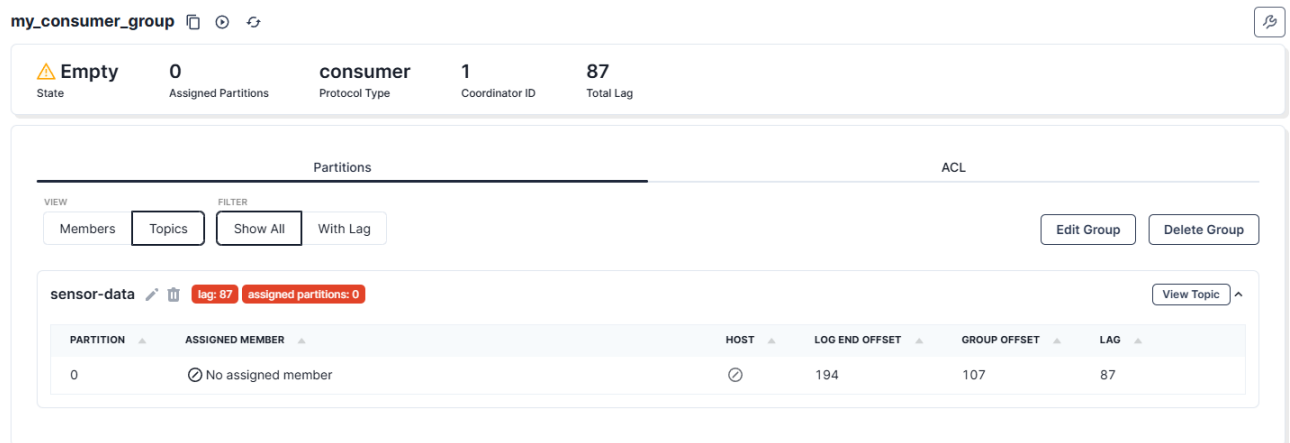


Figure 6.22 –Number of row consumed from Redpanda

Redpanda utilizes offset management to store the offset of the last consumed message for each consumer group. This mechanism allows a consumer to resume consumption from the last committed offset in case of a crash or restart, ensuring no data is lost or reprocessed. Furthermore, Redpanda replicates data across multiple nodes in the cluster, enabling the consumer to continue consuming from a replica without interruption in the event of a node failure.

Figure 3.22 illustrates how Redpanda manages offsets. The **LAG** value indicates that there are **87 messages** that have not yet been consumed by my_consumer_group and are waiting to be processed. So this meet the non-requirement for fault tolerance in **server side**.

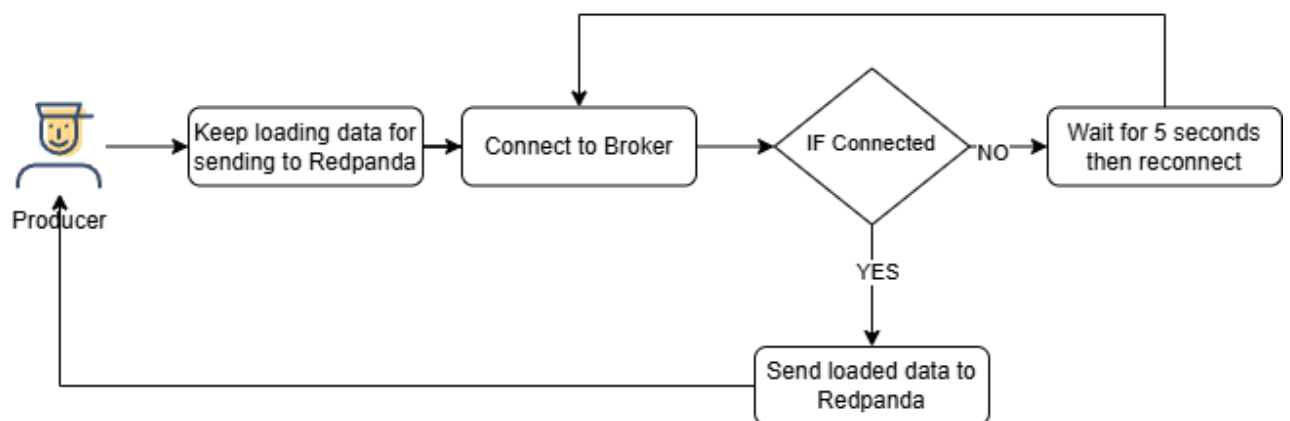


Figure 6.23 – Show the way handling disconnection in client side

Figure 3.23 illustrates the mechanisms employed to ensure seamless handling of disconnections on the client side, specifically for sensor nodes acting as producers. These

mechanisms are designed to maintain data integrity and ensure reliable communication with the Redpanda broker. Sensor nodes continuously attempt to reconnect to the Redpanda broker in case of a disconnection. Reconnection attempts follow a defined interval (5 seconds), preventing rapid, excessive retries that could overload the network or system. Once the connection is re-established, the producer immediately resumes data transmission, ensuring minimal disruption to the data flow.

Data generated by the sensor nodes but not yet transmitted to Redpanda during the disconnection is temporarily stored in a local buffer raw.txt file. The buffer acts as a persistent storage mechanism, ensuring no data is lost even during prolonged outages.

Monitoring and Manageability

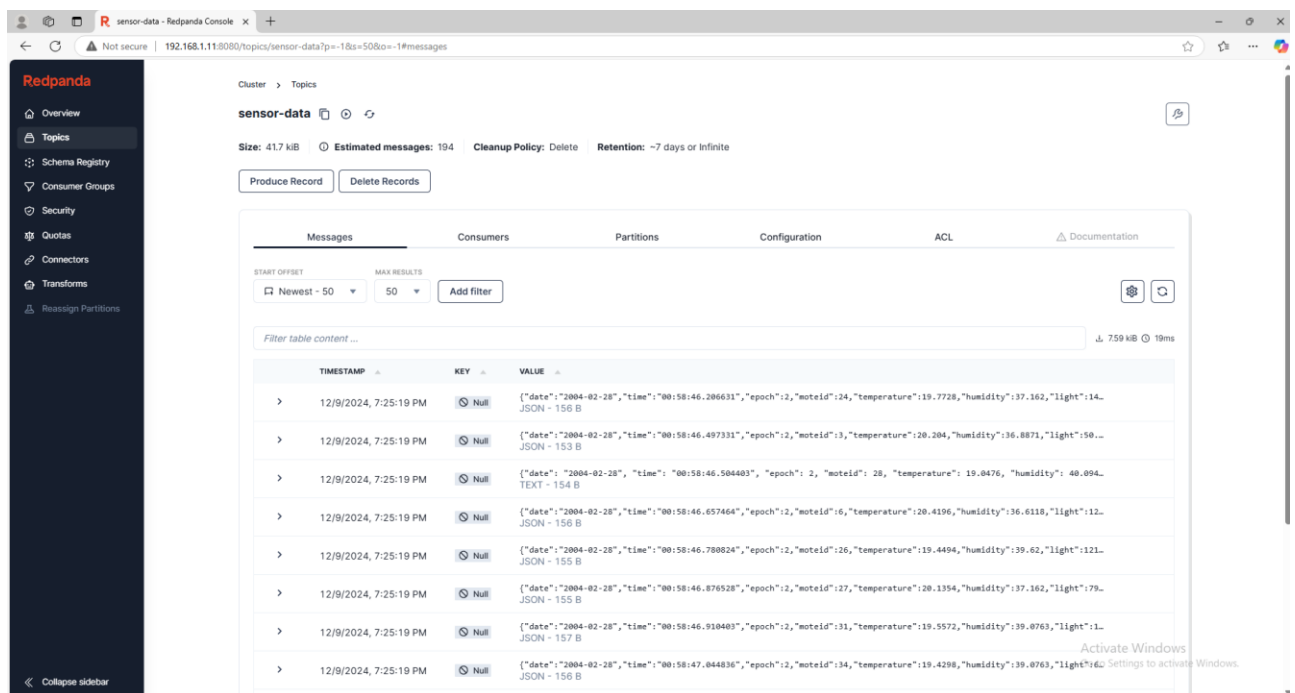


Figure 6.24 –Redpanda Console friendly UI

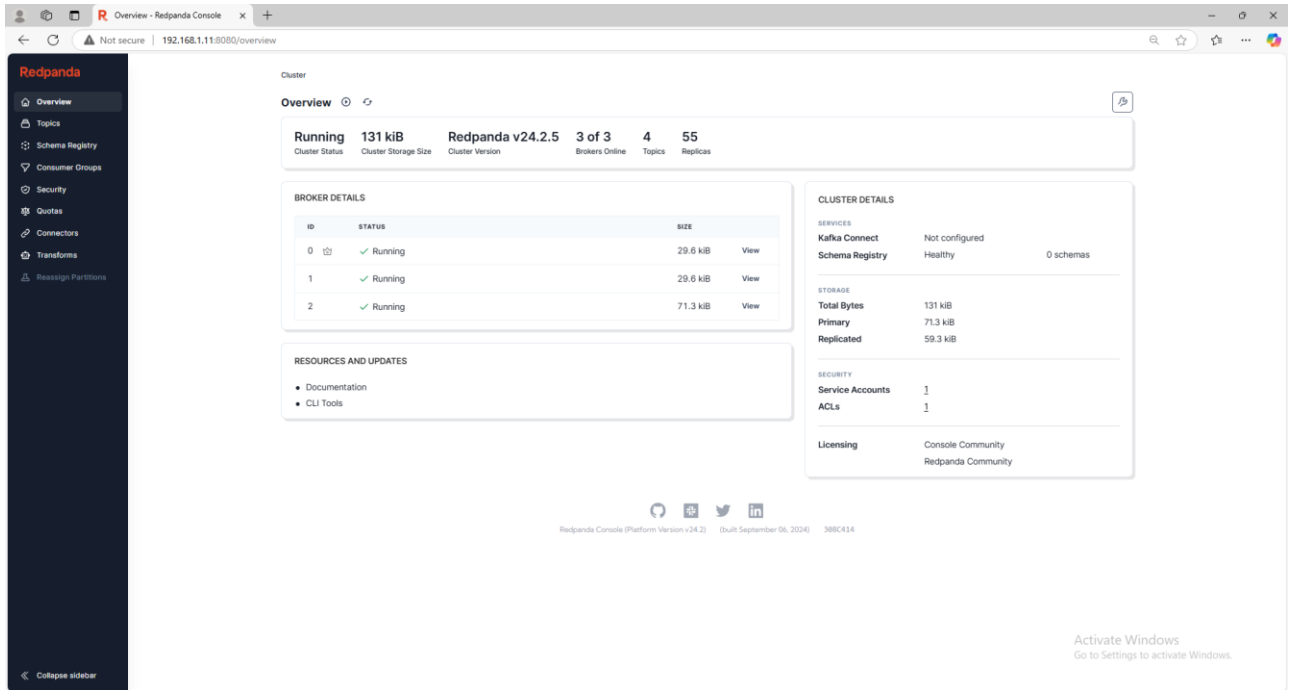


Figure 6.25 –Redpanda Broker Managing Page

This friendly UI console would meet the non-functional requirement for Monitoring and Manageability

6.2 GAMPS Compression

In this section, we present the results of our experimental evaluation of GAMPS, conducted on three real-world datasets: DataCenter (24 signals), IBT (54 signals), and Server (240 signals), as introduced at the beginning of Section 3. The evaluation is based on the implementation of the algorithm outlined in Algorithm 2.

6.2.1 Compression Performance

Our first experiment confirms that compression drastically reduces storage needs. We tested GAMPS on three datasets, dividing the approximation error into two parts. Adjusting this ratio had minimal effect on results but can be fine-tuned. As shown in Figure 6.1, GAMPS reduced storage by 100 to 490 times for the Server dataset and 25 to 125 times for the IBT and DataCenter datasets, even with small error tolerances (1–2%). The Server dataset performed best due to its size and larger average group size.

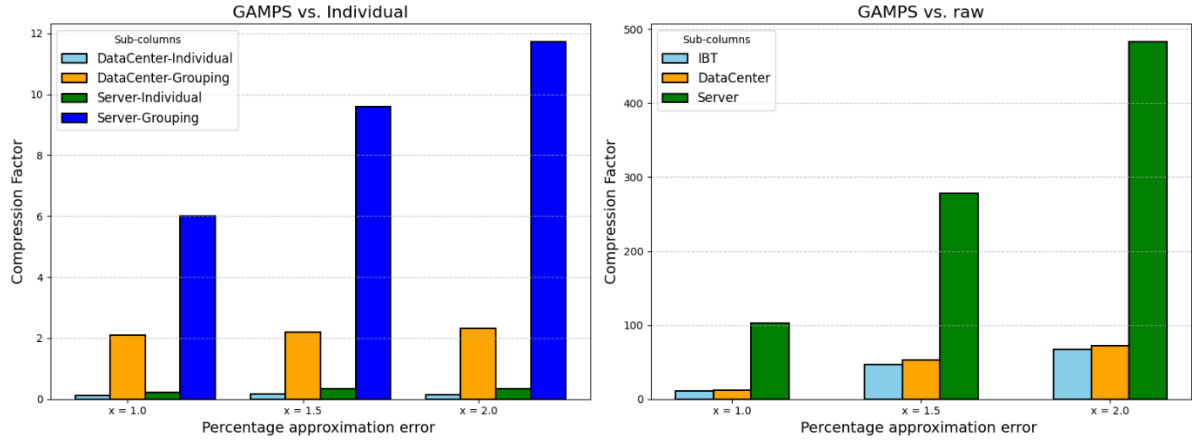


Figure 6.1 - Compression Efficiency Comparison of GAMPS

To compare GAMPS with other methods, we benchmarked it against a leading single-signal compression technique, illustrated in Figure 6.2. GAMPS provided 2 to 10 times more compression by leveraging inter-signal correlations at a 1.5% error tolerance. Notably, as error tolerances increased, GAMPS' advantage also grew, particularly for the Server dataset. This improvement stems from its ability to group correlated sensors, which is most effective with larger groups. The relationship between group size and compression is explored in the next experiment.

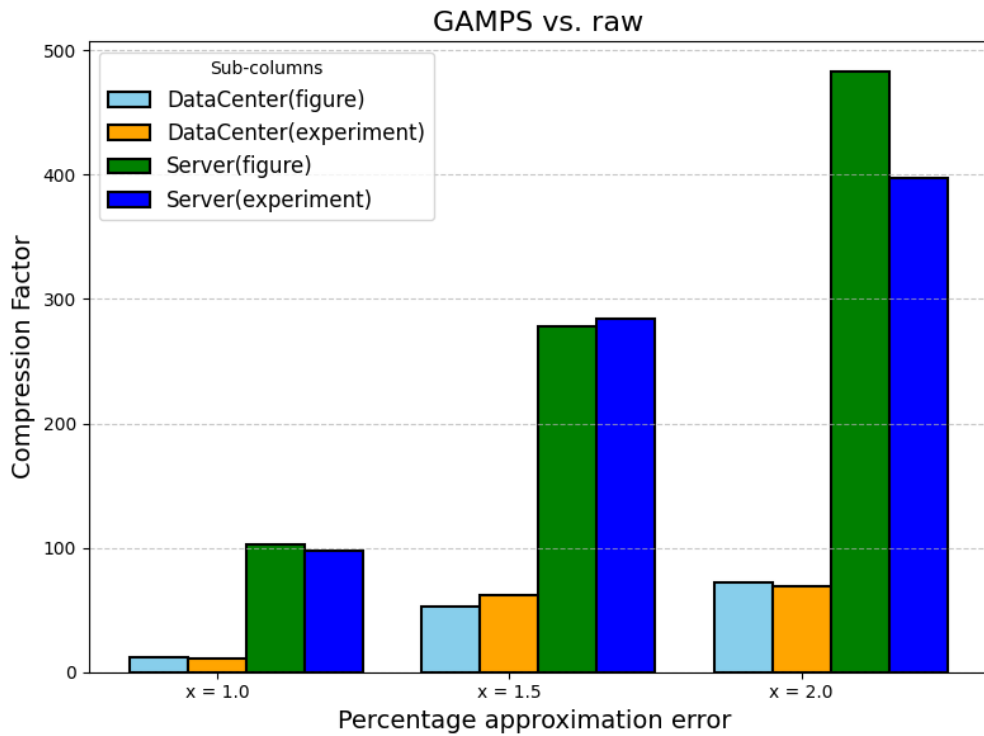


Figure 6.2 - GAMPS Compression Factor Comparison

6.2.2 Scaling with Group Size

In this experiment, we investigate how group size affects compression performance, using the 240-signal Server dataset, which is the largest in our study. We extracted a group of 60 signals from this dataset to examine the scaling of the compression factor with increasing group size. This setup follows the approach proposed in the paper by [7], where the performance of GAMPS is compared to individual signal compression using the bucketing scheme. As shown in figure, for the 60-signal Server data, GAMPS outperforms individual signal compression by a factor of 8 at $\varepsilon = 1\%$ and 12 at $\varepsilon = 2\%$, consistent with the settings proposed in the referenced paper.

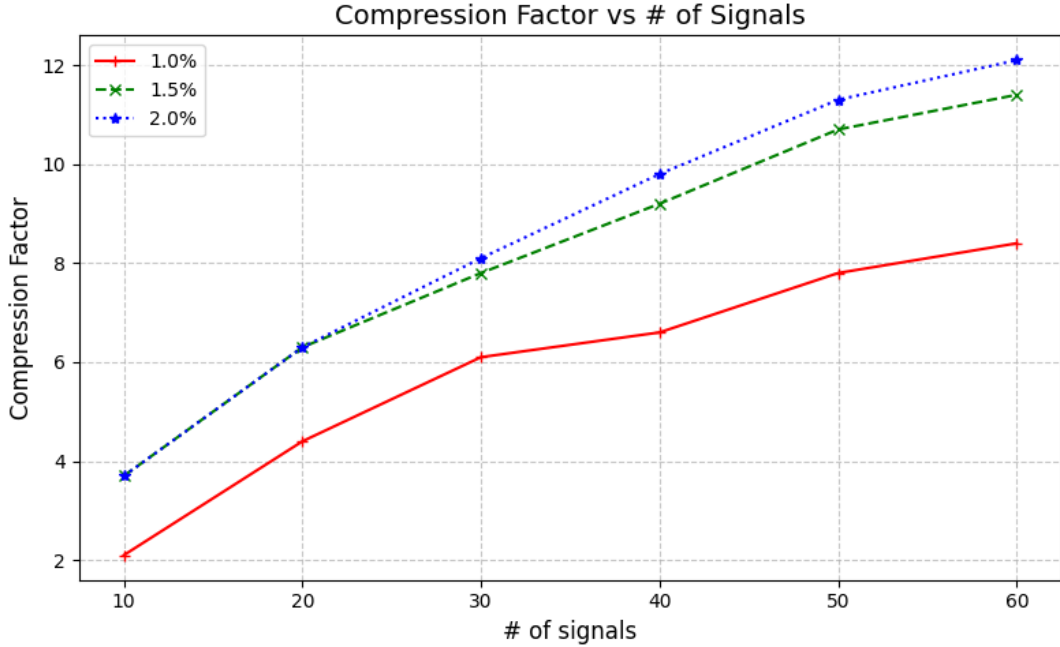


Figure 6.3 - Compression Factor vs Number of Signals

6.2.3 Compression Gain of Grouping

In this section, we empirically assess the importance of grouping in GAMPS by examining the space savings achieved through grouping, the impact of near-optimal grouping, and the advantages of dynamic grouping.

6.2.3.1 Static Grouping

To evaluate the significance of grouping, we first conducted a simple experiment where all the data was placed in a single group. We selected the best signal in the group as a

base, and the remaining signals were related to this base through ratios. However, we observed poor performance due to unrelated signals causing inaccurate ratio signals. We then introduced a hybrid approach, where a base signal is chosen for the group, but other signals can either join the group and use the ratio signal or maintain individual compression. The choice is based on the cost of representation. We expected the hybrid approach to perform at least as well as individual compression or a single group.

Figure 6.5 shows the relative compression achieved by individual compression, the hybrid approach, and GAMPS, using two datasets: the 45-signal IBT data and the 240-signal Server data. It is clear that a significant portion of the compression for both datasets comes from grouping. For example, for the Server dataset with a 1.5% error, the compression factor achieved by the hybrid approach over individual signal approximation is 1.5, compared to 9 achieved by GAMPS.

Interestingly, the grouping algorithm does not take into account the physical location of the sensors. It only calculates the cost of representing one signal with another and uses this information for grouping. Despite this, it is worth noting the geographical locality of sensors placed in the same groups. The left part of Figure 6.4 shows the sensor layout in the Intel Berkeley lab overlaid with the room layout, with sensors represented by dark hexagons. The right part shows the groups formed geographically, where sensors in each shaded region were grouped together by the algorithm. The simplicity of the group boundaries supports the intuition that sensors in the same group likely sense similar physical phenomena. This aligns with previous work by the authors, who mapped geographical regions to predict sensor values, noting that sensors in the same region were expected to have similar readings. Sensors marked with crosses lacked data for that day, and those marked with rectangles are outliers, where the grouping algorithm used individual approximations. Most of these outliers were located in or around conference rooms.

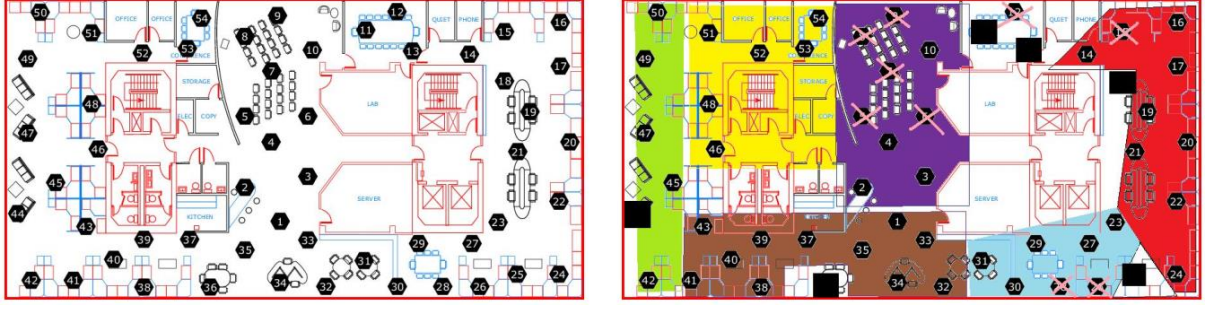


Figure 6.4 - The part on the left shows the sensor layout in the Intel Berkeley lab. The right part shows the groups found by the algorithm, along with outliers (marked by squares). [7]

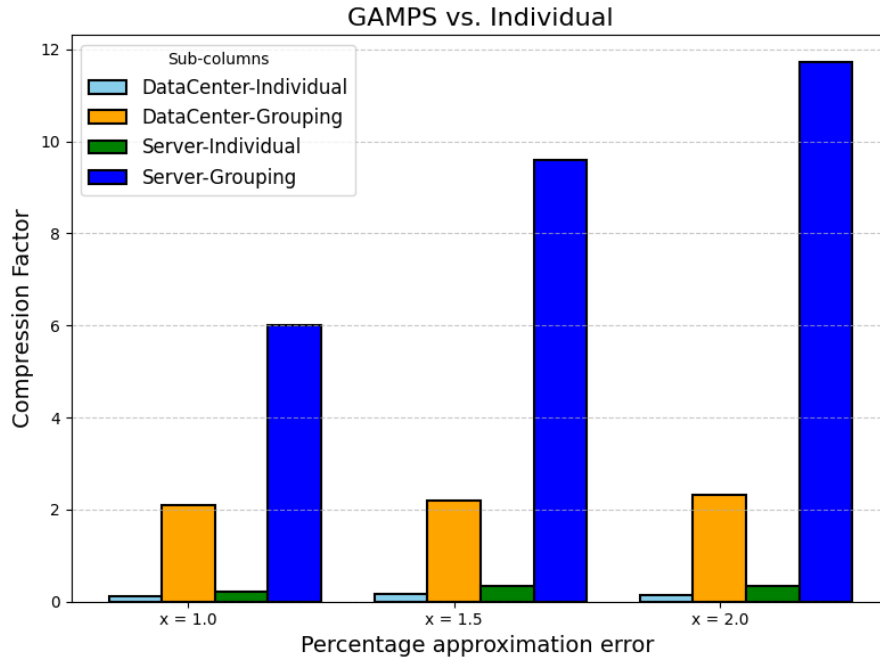


Figure 6.5 - relative compression achieved by individual compression, the hybrid approach, and GAMPS

6.2.3.2 Dynamic Grouping

Next, we evaluate the effectiveness of dynamic grouping compared to static grouping. Dynamic grouping has its trade-offs. Each time we regroup, additional information needs to be stored about the new groups, and non-full buckets at the group boundaries can lead to wasted space. However, if the regrouping interval is too long, we may miss out on compression benefits that could be achieved by adjusting the grouping. To assess these pros and cons, we used the DataCenter dataset for the experiment, as shown in Figure 6.6.

The key takeaway is that dynamic grouping consistently outperforms static grouping, though by a small margin. Additionally, our dynamic grouping scheme adapts to the data automatically, providing better performance than any of the fixed periodic regroupings we tested, with different frequencies (500, 1000, 2000, and 4000).

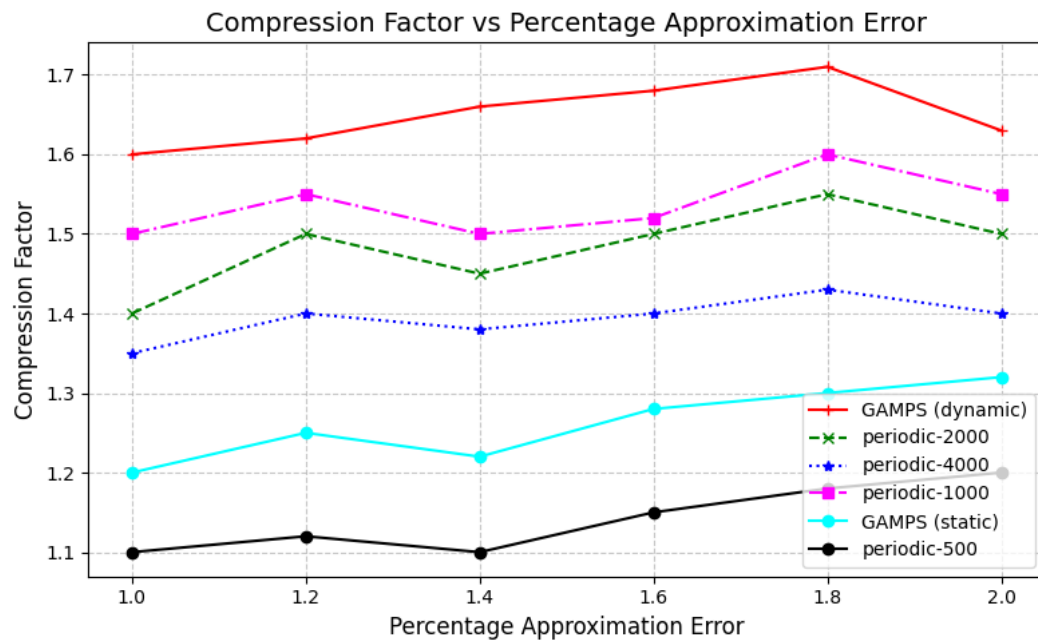


Figure 6.6 - Compression Factor vs Percentage Apporximation

Chapter 7 Conclusion

The deployment of an environmental monitoring system leveraging Redpanda for real-time data streaming and the GAMPS compression algorithm for data optimization demonstrated significant potential for scalable and efficient IoT applications. The project successfully addressed challenges associated with high-velocity and high-volume sensor data, ensuring minimal latency, effective data compression, and fault tolerance.

The key outcomes of this project include:

- **Effective Real-Time Streaming:** Using Redpanda, the system achieved low-latency, reliable streaming of environmental data, meeting the stringent requirements of real-time IoT applications.
- **Efficient Data Compression:** The GAMPS algorithm consistently outperformed traditional single-signal compression techniques, achieving compression factors up to 12x for large datasets, demonstrating the importance of grouping and amplitude scaling.
- **Dynamic Adaptability:** Dynamic grouping in GAMPS enhanced the adaptability of the system, allowing it to adjust compression strategies based on changing environmental conditions for sustained performance gains.
- **Scalability and Fault Tolerance:** The system's architecture supported robust handling of disconnections, offset management, and replication, ensuring continuous operation and data consistency.

These results showcase the feasibility of integrating modern streaming and compression technologies in IoT systems to optimize resource utilization, reduce costs, and maintain performance.

Bibliography

- [1] "What is Streaming Data? - Streaming Data Explained - AWS," Amazon Web Services, Inc., [Online]. Available: <https://aws.amazon.com/what-is/streaming-data/>. [Accessed 5 December 2024].
- [2] S. Sinha, "IoT Analytics," 3 September 2024. [Online]. Available: <https://iot-analytics.com/number-connected-iot-devices/>.
- [3] H. Dhaduk, "Stream Processing: How it Works, Use Cases & Popular Frameworks," Simform, 15 February 2023. [Online]. Available: <https://www.simform.com/blog/stream-processing/>. [Accessed 5 December 2024].
- [4] "Redpanda | The streaming data platform for developers," Redpanda, [Online]. Available: <https://www.redpanda.com/>. [Accessed 5 December 2024].
- [5] "What is Redpanda? | Redpanda," Redpanda, [Online]. Available: <https://www.redpanda.com/what-is-redpanda>. [Accessed 5 December 2024].
- [6] U. Jayasankar, V. Thirumal and D. Ponnurangam, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University – Computer and Information Sciences*, vol. 33, pp. 119-140, 2021.
- [7] S. Gandhi, S. Nath, S. Suri and J. Liu, "GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling," *SIGMOD*, pp. 771 - 784, 2009.