VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT

MULTIDISCIPLINARY PROJECT (CO3109)

Apache Pulsar & SprintZ Compression Application:

SMART HOME

SUPERVISOR(s):	Diệp Thanh Đăng, Ph.D.
STUDENT:	Phạm Lê Quân (CS) - 2153749
	Trương Huỳnh Anh Dũng (CS) - 2053622
	Nguyễn Văn Đức Long (CE) - 2153535
	Trần Nguyễn Gia Huy (CE) - 2153395

HO CHI MINH CITY, Dec 2024

Abstract

This project presents a multidisciplinary approach to integrating data streaming and compression systems tailored for IoT environments, where efficient data handling is paramount. By combining the Sprintz time-series compression algorithm with the Apache Pulsar messaging platform, the project aims to significantly reduce bandwidth and storage requirements. The primary objective is to develop a unified system where computer engineering (CE) students are responsible for configuring and deploying Apache Pulsar for data streaming between servers and clients, while computer science (CS) students focus on optimizing the Sprintz algorithm for high compression rates of IoT time-series data. This report provides a comprehensive overview of the system configuration, deployment processes, user guidance, evaluation methodologies, and additional technical insights that will support future enhancements and improvements.

Contents

List of Figures

Chapter 1

Introduction

In chapter 1, the overview, objectives and goals of the research's project are illustrated. The outline of the report is also presented.

1.1 Motivation

The increasing adoption of smart home systems has led to a surge in data generation from various sensors monitoring air quality, energy consumption, security, and other environmental parameters. However, this data presents a significant challenge in terms of scalability and efficiency, especially when transmitting large volumes of sensor data over networks. A common limitation in handling such data is the bandwidth and storage constraints on the servers processing this information.

Apache Pulsar offers an excellent solution as a distributed streaming platform capable of handling high-throughput, low-latency data streams. By leveraging Pulsar's capabilities for message queuing and real-time data distribution, along with the integration of Sprintz encoding, we aim to enhance data throughput, reduce server load, and optimize resource usage. Sprintz's compression ensures that data sent over the network is compressed before being published to topics, leading to better utilization of available bandwidth while ensuring timely data delivery.

1.2 Goals

Building upon this foundation, the project focuses on achieving key objectives that address the challenges of data efficiency, scalability, and system performance. These goals are designed to maximize the potential of Apache Pulsar and Sprintz compression, ensuring a seamless and optimized smart home data processing pipeline. Specifically, we aim to:

- **Optimize Data Transmission Efficiency**: By implementing Sprintz encoding in the producer, the project aims to reduce the size of the transmitted data, thus decreasing bandwidth usage and improving overall network performance.
- **Improve Throughput**: The goal is to ensure high throughput with minimal latency by streamlining the data flow between producers, Apache Pulsar, and consumers. Sprintz encoding will help compress the data, which will enable the system to handle more data with fewer resources.
- **Ensure Seamless Integration**: We aim to integrate Apache Pulsar and Sprintz smoothly with existing smart home devices, ensuring that the encoded data is efficiently consumed by applications while maintaining real-time capabilities.
- **Scalability and Reliability**: The system is designed to scale effortlessly with increasing numbers of sensors and data points, while maintaining reliable and robust data handling even under heavy loads.

1.3 Report structure

There are five chapters in this capstone project:

- **Chapter 1: Introduction**: This chapter provides the motivation, goals, and objectives for the project, emphasizing the need for scalable and efficient data processing in IoT environments.

- **Chapter 2: System Design & Technology**: This chapter explains the overall design and technical foundation of the system. It includes a detailed description of the Sprintz compression algorithm, Apache Pulsar's architecture, and key components such as tenants, namespaces, topics, brokers, ZooKeeper, and BookKeeper.

- **Chapter 3: System Implementation**: This chapter focuses on the practical aspects of implementing the system. It details the integration of Sprintz compression with Apache Pulsar and the design of the producer-consumer pipeline. Steps for compiling, integrating, and executing the system components are also covered.

- **Chapter 4: System Deployment & Demonstration**: This chapter outlines the procedures for deploying the system on Debian-based and Windows platforms. It provides step-by-step instructions for running producers and consumers and demonstrates the system's real-time data handling capabilities with accompanying visual results.

- **Chapter 5: Conclusion**: The final chapter summarizes the achievements of the project, discusses its contributions to IoT and smart home technologies, and outlines potential areas for future work, such as enhancing compression algorithms, expanding use cases, and improving system scalability and reliability.

Chapter 2

System Design & Technology

2.1 Overall

2.1.1 System Design

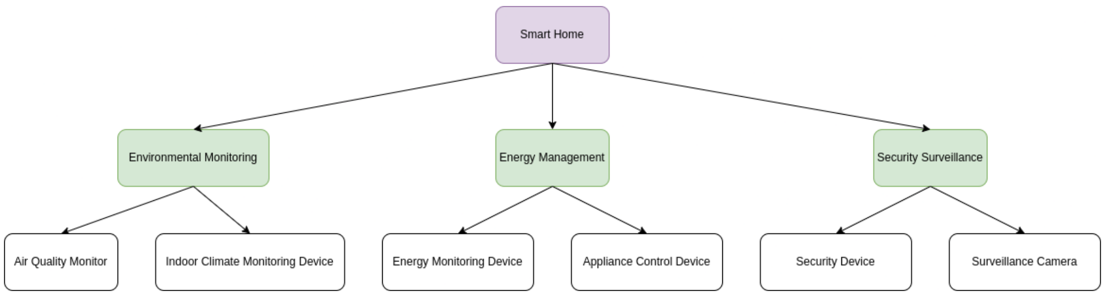


Figure 2.1: System Design

The system design of our Smart Home project is built to classify and manage various types of sensor devices efficiently. The structure is divided into three main categories to provide better organization and scalability:

- **Environmental Monitoring**: This category encompasses sensors like the Air Quality Monitor and Indoor Climate Monitoring Device, which focus on monitoring environmental parameters such as air quality and indoor climate conditions.

- **Energy Management**: This section includes sensors like the Energy Monitoring Device and Appliance Control Device, which are responsible for monitoring and controlling energy usage and appliances.

- **Security Surveillance**: This category is dedicated to ensuring safety and includes devices like Security Device and Surveillance Camera for monitoring security threats and maintaining surveillance.

The primary motivation behind this hierarchical structure is to create a modular and easily expandable system. By grouping sensors into specific fields (Environmental Monitoring, Energy Management, and Security Surveillance), the system can scale effortlessly with new devices or technologies without disrupting existing configurations.

This design ensures the following:

- **Scalability**: New devices or sensors can be seamlessly integrated into the system by adding them under their respective categories.
- **Efficiency**: Organized topics improve data handling within Apache Pulsar, making it easier to process and consume data.
- **Flexibility**: The modular design supports modifications or expansions without the need for significant overhauls.

This thoughtful approach ensures that the Smart Home system is not only efficient and scalable but also adaptable to future advancements in smart home technologies.

2.1.2 Data Compression

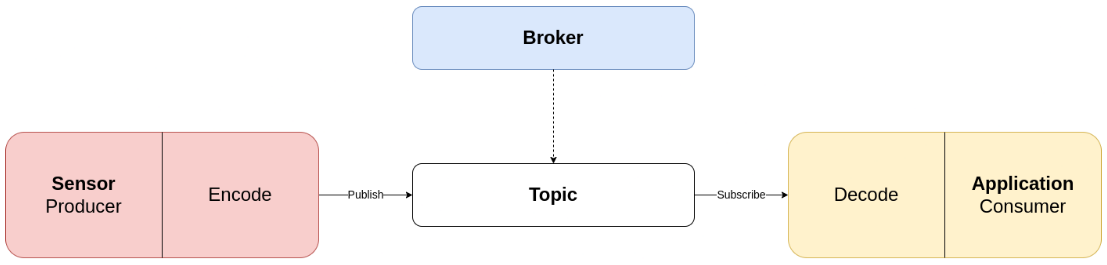


Figure 2.2: Data Compression in our system

How Compression Impacts Throughput

1. **Compression on the Producer Side (Sensor to Topic)**: Before the sensor data is published to a topic, it is encoded using a compression algorithm.
   Impact: Compression reduces the size of the data being transmitted to the broker and stored in the topic. This minimizes the bandwidth required for transmission and reduces the load on the network, leading to increased throughput in terms of the volume of messages the system can handle per unit of time. However, compression introduces an overhead in terms of computational resources and time. If the encoding process is computationally expensive, it could slightly delay the publishing of data, which may reduce throughput if the producer is operating at its computational limit.

2. **Transmission Through the Broker**: Compressed data requires less bandwidth for transmission between the broker and topic storage. This means the broker can handle more messages concurrently, as smaller data sizes reduce network latency and storage load.
   Impact: The system's throughput can increase as the broker can process and store more data efficiently without becoming a bottleneck.

3. **Decompression on the Consumer Side (Topic to Application)**: When data is consumed by the application, it needs to be decoded (decompressed). This process requires computational effort on the consumer side.

Impact: If the decompression algorithm is not optimized or if the consumer application has limited computational resources, this additional step can decrease throughput, as the consumer might take longer to process the messages.

Overall Considerations

- **Compression Algorithm Efficiency**: A well-optimized compression algorithm can minimize the computational overhead and enhance overall throughput.
- **Trade-off Between Size and Speed**: Smaller compressed sizes reduce network load but may increase computation time. The system design must balance these factors to achieve optimal throughput.
- **Scalability**: For systems with high message volumes across multiple topics, compression greatly improves scalability, allowing more topics and data streams to coexist without overloading the infrastructure.

By integrating compression into the data pipeline (producer $\rightarrow$ topic $\rightarrow$ consumer), the system achieves better network and storage efficiency. However, the computational overhead introduced by the compression and decompression processes must be carefully managed to avoid bottlenecks at the producer or consumer stages.

2.2 Compression Technology: Sprintz

Sprintz is a cutting-edge compression algorithm tailored for time-series data. It achieves state-of-the-art compression ratios while meeting the stringent requirements of low memory usage and minimal latency, making it ideal for resource-constrained environments[2].

Key Features of Sprintz

- **High Efficiency:** Sprintz ensures fast compression and decompression speeds, capable of decompressing data at over 3 GB/s on a single thread.
- **Minimal Resource Footprint:**
  - Operates with less than 1 KB of memory, making it suitable for devices with limited resources.
  - Utilizes blocks as small as eight samples, significantly reducing latency for continuously arriving data.
- **Lossless Compression:** The algorithm guarantees the integrity of the data, preserving every detail for accurate downstream analysis.
- **Advanced Forecasting:** Sprintz incorporates vectorized forecasting methods such as delta coding and FIRE (Fast Integer Regression), which leverage temporal correlations for improved compression performance.
- **Innovative Techniques:** Employs techniques like bit-packing, run-length encoding, and entropy coding to maximize compression ratios while maintaining high throughput.
- **Scalability:** Its design supports multivariate time-series data, enabling efficient processing of datasets with multiple variables and increasing dimensions.

By exploiting temporal correlations, leveraging efficient predictive algorithms, and incorporating vectorized operations, Sprintz delivers exceptional performance in compressing time-series data, combining high speed with robust compression ratios and low resource requirements.

2.3 Data Streaming Technology: Apache Pulsar

2.3.1 Introduction

Apache Pulsar is a distributed messaging and streaming platform that enables real-time data transfer and processing for modern applications. It was originally developed by Yahoo to handle massive data flows with low latency, and in 2016, it was open-sourced and later adopted by the Apache Software Foundation.[3] Today, it is recognized as a robust solution for organizations needing highly scalable, fault-tolerant, and high-performance messaging systems.

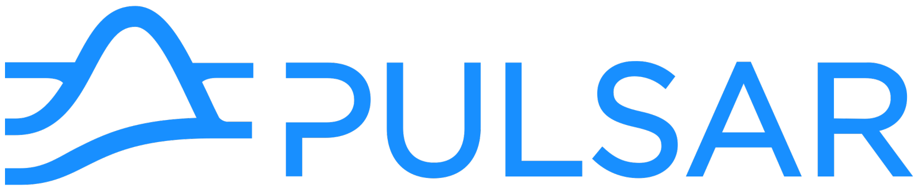


Figure 2.3: Apache Pulsar Logo [4]

2.3.1.1 Publish-Subscribe Model

At its core, Apache Pulsar operates on a publish-subscribe model, where producers publish messages to topics, and consumers subscribe to these topics to process the messages. This design allows for seamless, real-time communication between various components of a distributed system.

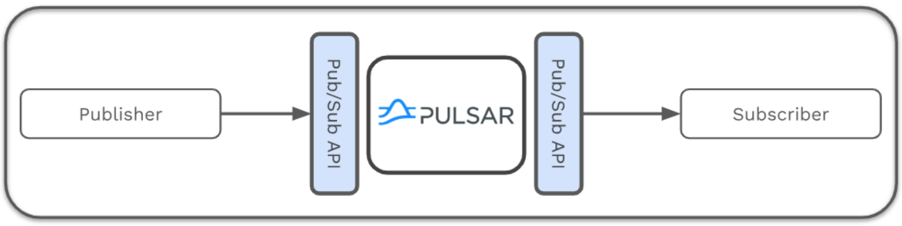


Figure 2.4: Publish-Subribe model [5]

This model decouples the producers, brokers, and consumers, allowing for scalable and flexible communication across distributed systems:

- Producers: Entities that publish messages to specific topics. For instance, an IoT device might act as a producer, sending temperature readings to a topic.

- Topics: Logical channels where messages are published and organized. Topics act as a bridge between producers and consumers.
- Consumers: Entities that subscribe to topics to receive and process messages. Consumers can process data in real time or store it for analysis.

This model ensures seamless, real-time communication while maintaining scalability and fault tolerance.

2.3.1.2 Tenants, Namespaces, and Topics

Apache Pulsar introduces a hierarchical data structure to organize and isolate workloads efficiently, Tenants, Namespaces, and Topics:

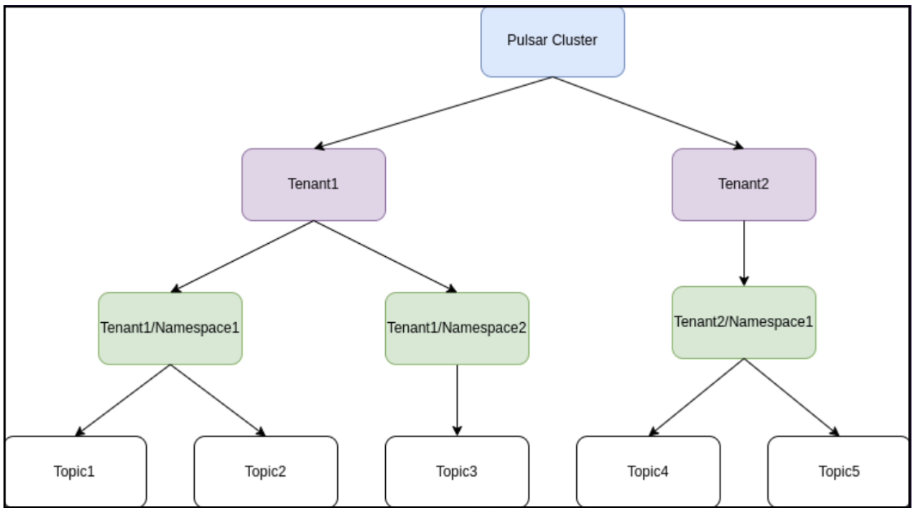


Figure 2.5: Structure [1]

1. **Tenants:** represent isolated groups in a Pulsar instance. Each tenant can have its own set of namespaces and policies, enabling multiple teams or organizations to share the same infrastructure securely
2. **Namespaces:** are logical groupings under a tenant, organizing related topics and managing configurations. It is useful for isolating environments (like different departments in a company) or grouping similar data streams.

3. **Topics:** are the smallest units of organization in Pulsar. Producers publish messages to topics, and consumers subscribe to them to process the data. Pulsar supports two types of topics:

   - Persistent Topics: Messages are stored durably in Apache BookKeeper.
   - Non-Persistent Topics: Messages are delivered in memory for ultra-low-latency scenarios

2.3.1.3 Subscription Types

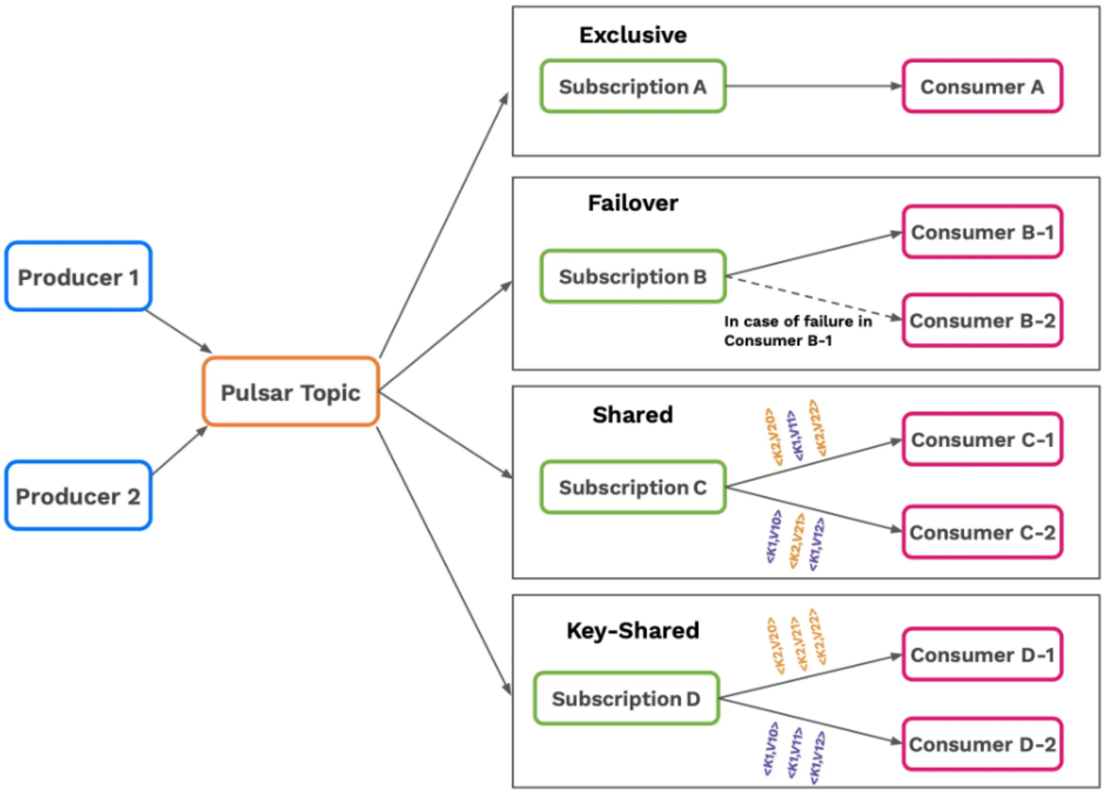


Figure 2.6: Subscription Types [6]

Pulsar's flexibility comes from its subscription types, which dictate how consumers receive messages. The four types are:

1. **Exclusive:** Only one consumer can attach to the subscription. Ensures strict message order.

2. **Failover:** Multiple consumers can attach, but only one is active at a time. If

the active consumer fails, the next one in line takes over.

3. **Shared:** Messages are distributed among all active consumers. Offers high throughput but no guaranteed order.

4. **Key-Shared:** Similar to Shared, but maintains order for messages with the same key.

Why Apache Pulsar?

In the era of Big Data and Internet of Things (IoT), organizations require messaging systems that can process millions of events per second, handle geographically distributed setups, and ensure data durability. Apache Pulsar addresses these challenges with features like geo-replication, persistent storage, and scalable architecture.

Unlike traditional messaging systems, Apache Pulsar bridges the gap between messaging and real-time data streaming. It not only ensures reliable message delivery but also supports features like serverless functions through Pulsar Functions, enabling lightweight computation within the system.

Use Cases

Apache Pulsar is widely adopted across various industries, including:

1. IoT: Real-time monitoring and data ingestion from devices like sensors and cameras.

2. Financial Services: Handling high-speed transactions and fraud detection systems.

3. Real-Time Analytics: Powering dashboards and data pipelines for decision-making.

4. E-Commerce: Managing order processing, notifications, and user activity logs.

With its modular architecture and extensive feature set, Apache Pulsar has become a preferred choice for organizations looking for a next-generation messaging and streaming solution.

2.3.2 Architecture

Overview

Apache Pulsar is designed as a highly scalable and fault-tolerant messaging and streaming platform. Its architecture is organized into instances, clusters, and key components like brokers, ZooKeeper, and BookKeeper, each playing a specific role in ensuring seamless data flow, coordination, and persistent storage. This modular design allows Pulsar to scale horizontally while maintaining reliability and performance.

2.3.2.1 Instance and Clusters

At the highest level, a Pulsar instance comprises one or more clusters. Clusters within the same instance can replicate data between themselves, enabling geo-replication for distributed systems. Each Pulsar cluster includes the following components:

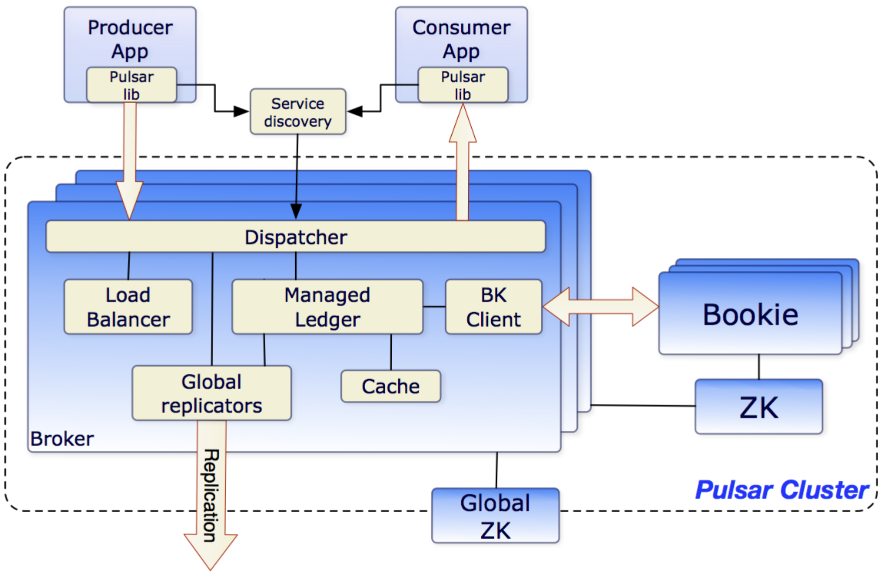


Figure 2.7: Pulsar cluster[3]

- Brokers: Manage message flow, load balancing, and coordination with ZooKeeper and BookKeeper.
- ZooKeeper Quorum: Handles metadata storage and cluster-specific coordination.
- BookKeeper Ensemble: Provides persistent storage for messages using a distributed log architecture.

2.3.2.2 Brokers

Overview

Brokers in Apache Pulsar serve as the stateless serving layer responsible for managing the interactions between producers, consumers, and the distributed storage system. Their primary roles include handling client connections, managing message routing, and distributing messages to appropriate topics. This separation of concerns ensures high scalability and fault tolerance in distributed messaging environments.

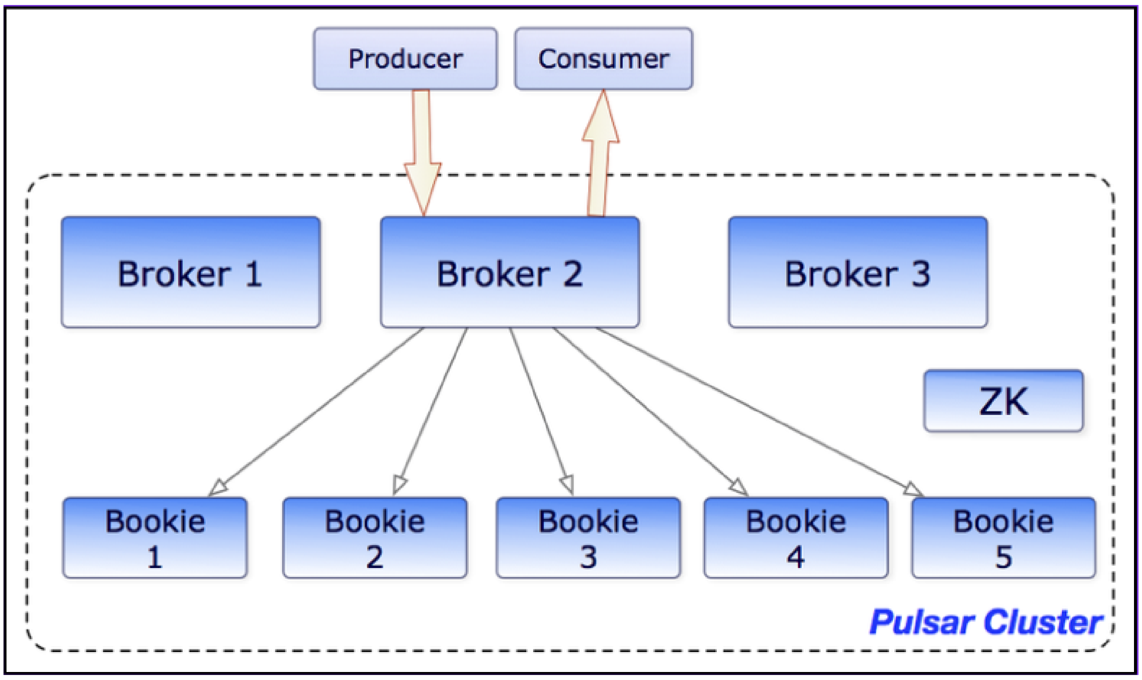


Figure 2.8: Broker diagram [7]

Unlike traditional messaging systems where brokers may store messages temporarily, Apache Pulsar brokers do not retain any messages locally. Instead, all messages are persisted in Apache BookKeeper, a separate storage layer, which offloads storage concerns from the broker. This design ensures that brokers remain lightweight and can scale independently.

How Brokers Work in Apache Pulsar:

1. Topic Partition Assignment: Each topic partition in Pulsar is assigned to a specific broker. Producers and consumers connect to the broker hosting the relevant partition to send or retrieve messages. This mapping of partitions to brokers allows for efficient message routing.

2. Message Routing: Brokers act as intermediaries, routing messages from producers to consumers. They ensure that messages reach their intended destinations without the need for direct communication between producers and consumers.

3. Fault Tolerance:In case of a broker failure, the ownership of the affected topic partitions is transferred seamlessly to another available broker. This process does not require moving the data itself because messages are stored in BookKeeper, making the failover process fast and reliable.

4. Client Connection Management: Brokers handle client connections for both producers and consumers, maintaining communication channels and ensuring smooth message exchanges.

5. High Availability: Pulsar brokers are designed to operate in clusters, ensuring that the system remains available even if individual brokers experience issues. This cluster-based approach is crucial for applications requiring continuous uptime.

Apache Pulsar brokers play a pivotal role in managing the flow of data between producers, consumers, and the storage layer. Their stateless design, fault tolerance, and efficient routing mechanisms make them essential for building scalable, resilient, and high-performance messaging systems.

Whether handling real-time IoT data, powering analytics pipelines, or ensuring reliable communication across distributed systems, brokers provide the backbone for Apache Pulsar's robust architecture.

2.3.2.3 ZooKeeper

In Apache Pulsar, Apache ZooKeeper plays a crucial role in managing configuration and coordination tasks within the cluster. ZooKeeper is an open-source project that provides distributed configuration services, synchronization, and naming registries for large-scale distributed systems.

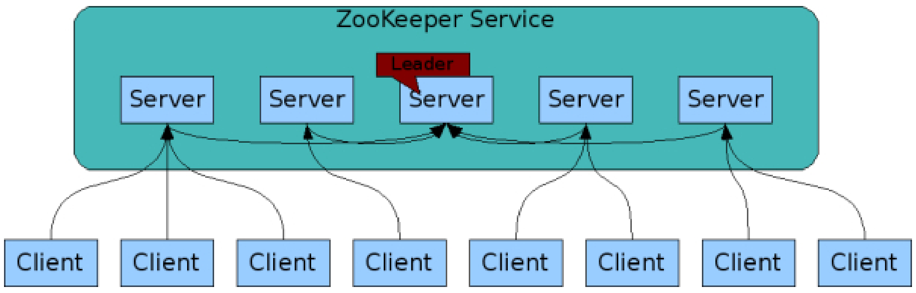


Figure 2.9: Zookeeper diagram [8]

Key Responsibilities of ZooKeeper in Pulsar:

- Configuration Management:

  ZooKeeper stores essential configuration data for Pulsar clusters, including details about brokers, topics, tenants, and namespaces. This centralized configuration ensures consistency across the cluster.

- Metadata Storage:

  It maintains metadata about topics, such as the mapping of topics to brokers and the status of each topic partition. This information is vital for efficient message routing and delivery.

- Leader Election:

  ZooKeeper facilitates the leader election process among brokers, ensuring that a designated leader manages specific tasks like assigning topics to brokers and handling failovers. This mechanism enhances the resilience and reliability of the Pulsar cluster.

- Cluster Coordination:

  It keeps track of the health and status of each broker using ephemeral znodes, which exist as long as the corresponding broker session is active. If a broker becomes unresponsive, ZooKeeper detects this change and helps reassign tasks to other active brokers, maintaining the cluster's stability.

By leveraging ZooKeeper for these critical functions, Apache Pulsar ensures robust coordination, consistent configuration management, and high availability within its distributed messaging and streaming platform.

2.3.2.4 BookKeeper

Overview

Apache BookKeeper is the persistence layer for Apache Pulsar, responsible for storing and managing messages across multiple storage units known as Bookies. Its primary goal is to ensure durable storage, scalability, and fault tolerance for all messages exchanged within the Pulsar ecosystem. By offloading storage responsibilities from brokers, BookKeeper allows Apache Pulsar to maintain its stateless design and ensures that data is reliably stored, even in the event of failures.

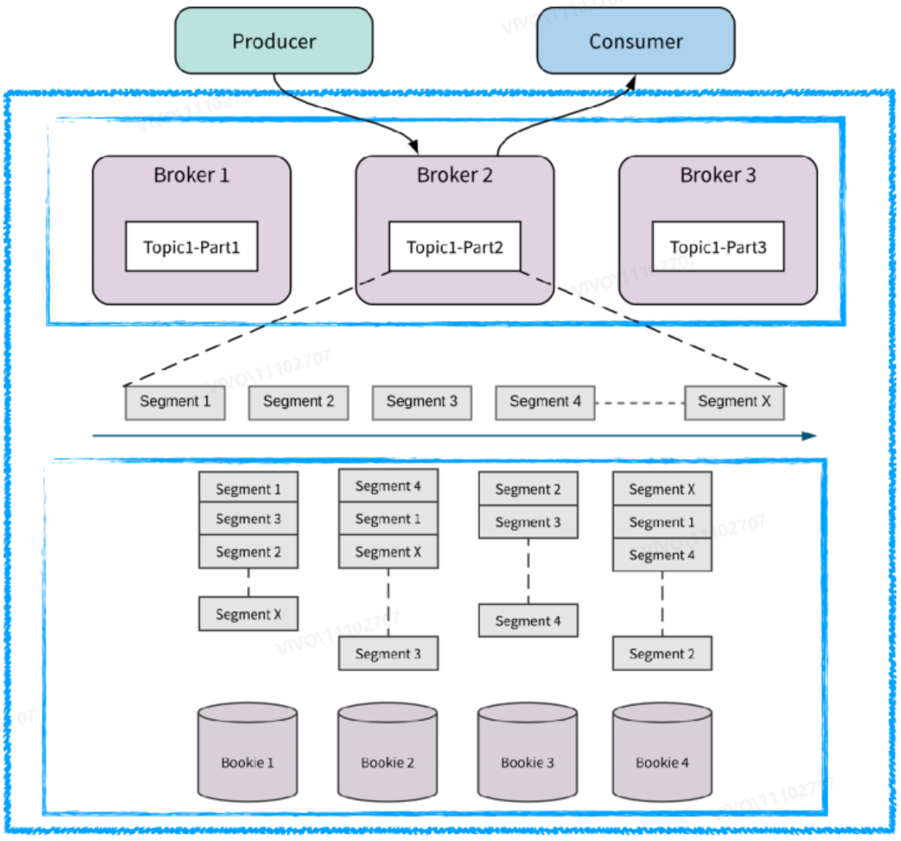


Figure 2.10: Bookkeeper diagram [9]

How BookKeeper Works

1. Distributed Logs: BookKeeper organizes stored data as distributed logs. Each topic partition in Pulsar is stored as a distributed log, ensuring efficient write and read operations across multiple storage nodes.

2. Segments: Each distributed log is divided into smaller units called segments. These segments:

   - Are written sequentially to disk.
   - Are replicated across multiple Bookies to ensure data redundancy and fault tolerance.
   - Have configurable limits based on size, time, or ownership changes. This segmentation ensures scalability by allowing new data to be stored without overloading any single storage unit.

3. Replication: To enhance reliability, each segment is replicated across a set of Bookies. If one Bookie fails, the replicated data ensures no loss of information, as the data can be retrieved from other Bookies in the cluster.

4. Dynamic Segment Management: New segments are automatically created as the log grows. This approach eliminates the need for static size configurations and supports uninterrupted data flow.

5. Independent from Brokers: Because data storage is entirely managed by BookKeeper, brokers can focus solely on routing messages and managing client connections. This separation of responsibilities simplifies the architecture and enhances performance.

Apache BookKeeper is a critical component of Pulsar's architecture, offering a robust and scalable solution for message persistence. Its ability to manage distributed logs, ensure data redundancy, and dynamically segment storage makes it an essential backbone for Pulsar's messaging and streaming capabilities. Whether handling IoT data, financial systems, or real-time analytics, BookKeeper ensures that messages are stored reliably and accessible when needed.

2.3.3 Key concepts

Key features of Pulsar are listed below:

- Native support for multiple clusters in a Pulsar instance, with seamless geo-replication of messages across clusters.
- Very low publish and end-to-end latency.
- Seamless scalability to over a million topics.
- A simple client API with bindings for Java, Go, Python and C++.
- Multiple subscription types (exclusive, shared, and failover) for topics.

- Guaranteed message delivery with persistent message storage provided by Apache BookKeeper. A serverless lightweight computing framework Pulsar Functions offers the capability for stream-native data processing.

- A serverless connector framework Pulsar IO, which is built on Pulsar Functions, makes it easier to move data in and out of Apache Pulsar.

- Tiered Storage offloads data from hot/warm storage to cold/long-term storage (such as S3 and GCS) when the data is aging out.

Chapter 3

System Implementation

3.1 Sprintz

3.1.1 Main Components of Sprintz

SprintZ is built on four main components, each of which plays an important role in optimizing the data compression process[2]:

- **Forecasting:** Sprintz uses prediction algorithms such as delta coding to predict the next value in the time series.
- **Bit Packing:**
  - After prediction, Sprintz stores the difference between the predicted value and the actual value in a "bit packing" format.
  - **Optimizing storage space:** This technique ensures that only the minimum number of bits required to represent the data is used. This is especially effective for low-variability data, where the differences are usually small and require few bits to encode.
  - **Compressed data structure:** Each data block is organized so that it is easy to retrieve and decompress, ensuring fast processing speed.
- **Run-Length Encoding (RLE):**

  - In cases where the data is unchanged or the difference values are zero in many consecutive blocks, Sprintz uses RLE to record the number of consecutive blocks instead of storing the entire data.
  - **Efficiency of RLE:** This technique helps achieve extremely high compression ratios in cases where the data is stable, such as a constant ambient temperature or a sensor device in an idle state.

- **Entropy Coding:**
  - After compressing with bit packing and RLE, Sprintz uses Huffman coding to further optimize the compressed data.
  - **The role of Huffman coding:** This algorithm compresses data based on the frequency of occurrence of characters, which helps reduce the size of compressed data without significantly increasing the complexity of decompression.

3.1.2 Sprintz Operation

3.1.2.1 Compression Process:

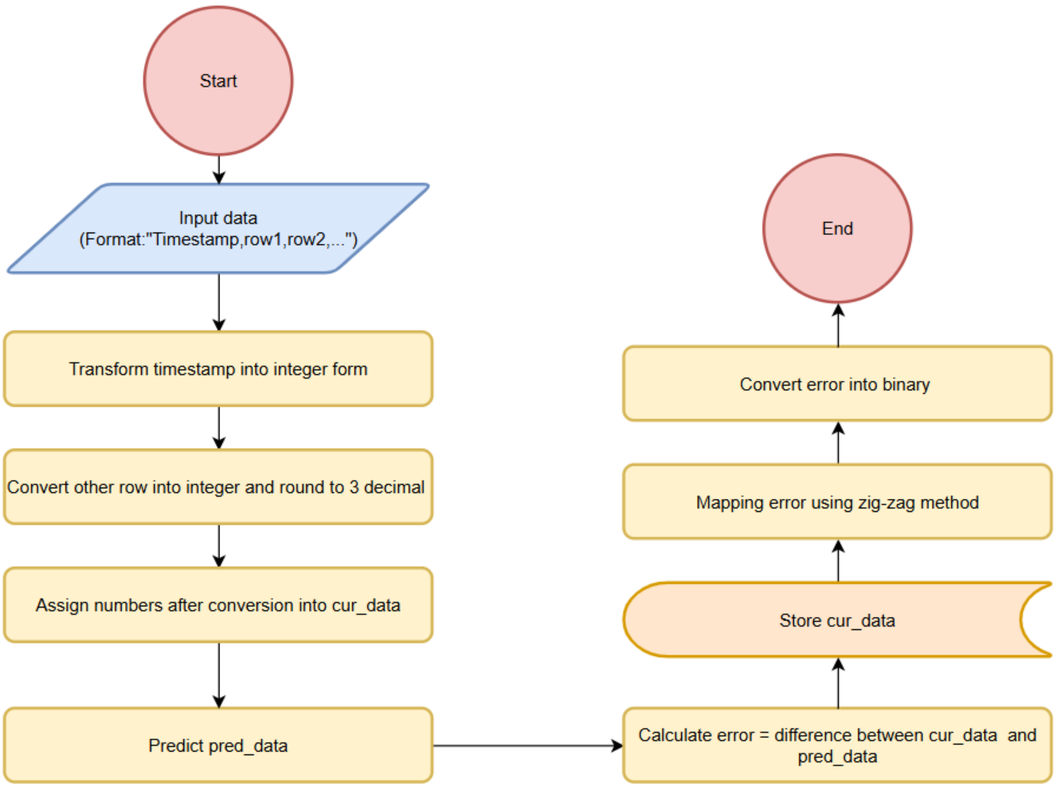

Start
Input data
(Format:"Timestamp,row1,row2,...")
Transform timestamp into integer form
Convert other row into integer and round to 3 decimal
Assign numbers after conversion into cur_data
Predict pred_data
Calculate error = difference between cur_data and pred_data
Store cur_data
Mapping error using zig-zag method
Convert error into binary
End

Figure 3.1: Encryption algorithm diagram

3.1.2.2 Decompression Process:

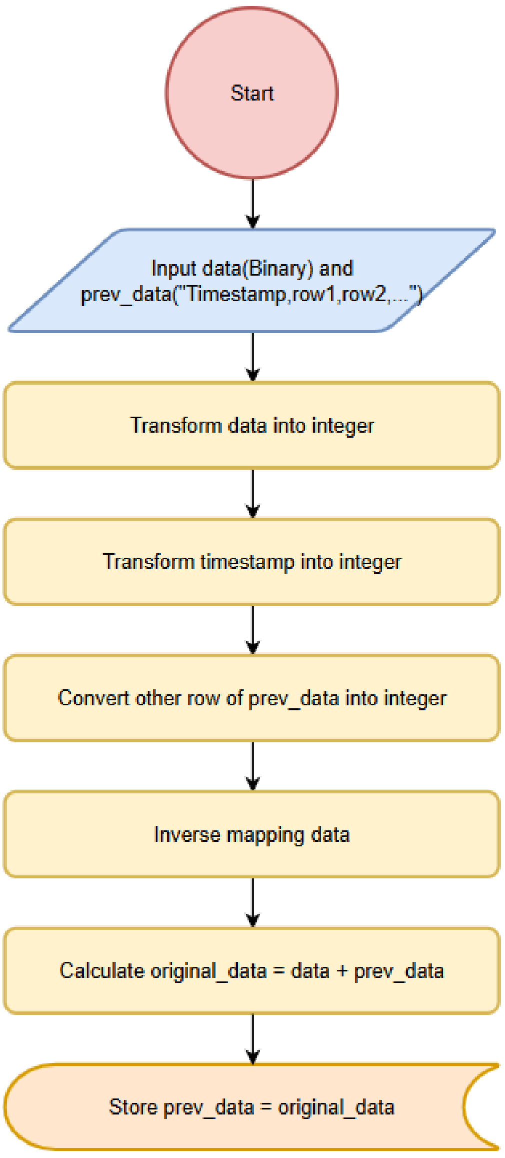
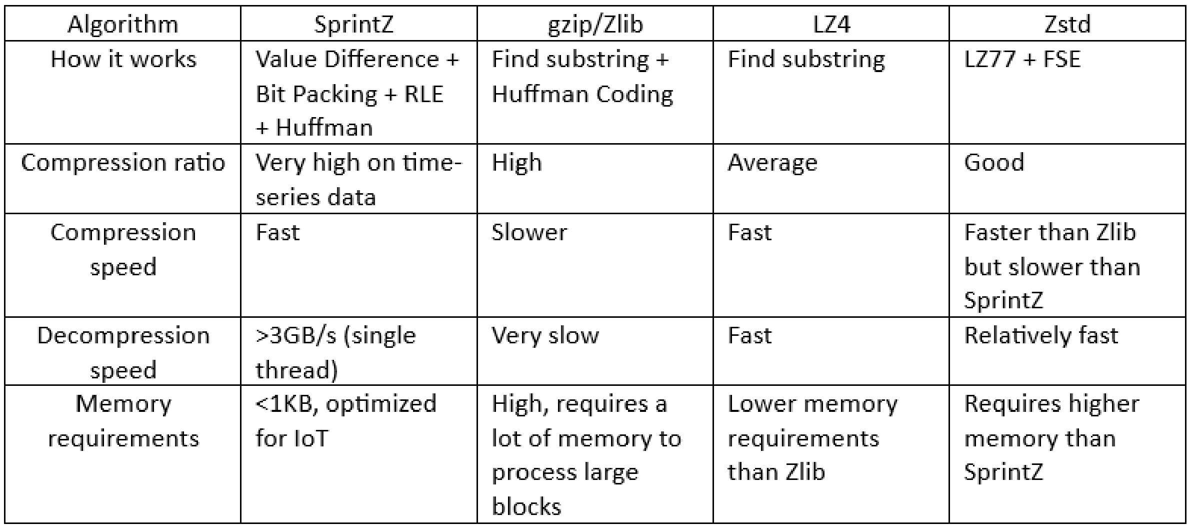

Start
Input data(Binary) and prev_data("Timestamp,row1,row2,...")
Transform data into integer
Transform timestamp into integer
Convert other row of prev_data into integer
Inverse mapping data
Calculate original_data = data + prev_data
Store prev_data = original_data

Figure 3.2: Decoding algorithm diagram

3.1.3 Comparison

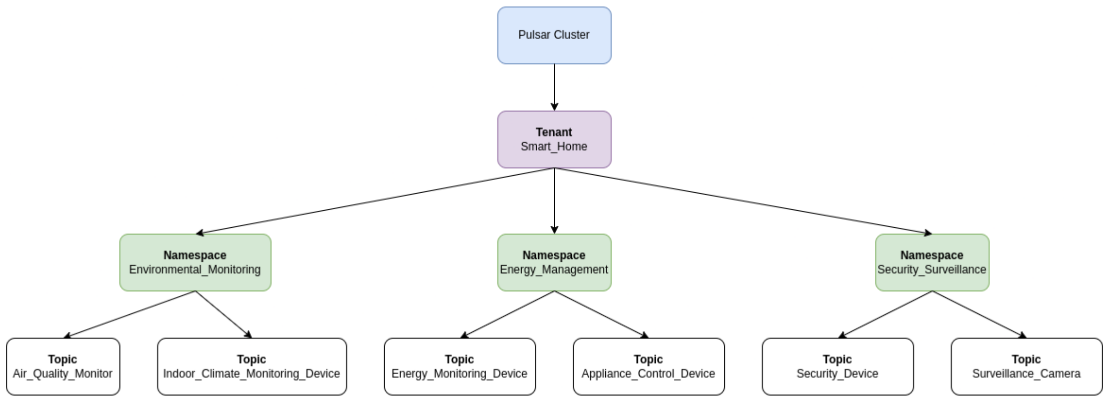


Algorithm	SprintZ	gzip/Zlib	LZ4	Zstd
How it works	Value Difference + Bit Packing + RLE + Huffman	Find substring + Huffman Coding	Find substring	LZ77 + FSE
Compression ratio	Very high on time-series data	High	Average	Good
Compression speed	Fast	Slower	Fast	Faster than Zlib but slower than SprintZ
Decompression speed	>3GB/s (single thread)	Very slow	Fast	Relatively fast
Memory requirements	<1KB, optimized for IoT	High, requires a lot of memory to process large blocks	Lower memory requirements than Zlib	Requires higher memory than SprintZ

Figure 3.3: Comparison with other algorithm

3.2 Apache Pulsar

3.2.1 Overall Pulsar Architecture



Pulsar Cluster
Tenant Smart_Home
Namespace Environmental_Monitoring
Namespace Energy_Management
Namespace Security_Surveillance
Topic Air_Quality_Monitor
Topic Indoor_Climate_Monitoring_Device
Topic Energy_Monitoring_Device
Topic Appliance_Control_Device
Topic Security_Device
Topic Surveillance_Camera

Figure 3.4: Pulsar Architecture

The architecture of Apache Pulsar is structured around a hierarchical design to handle data streams efficiently and ensure seamless scalability and modularity in data management. The core components of this architecture include the **Pulsar Cluster**, **Tenant**, **Namespaces**, and **Topics**, which are organized to classify and manage sensor data effectively.

- **Pulsar Cluster**: At the top level of the architecture is the Pulsar Cluster, which acts as the central data management system. It is responsible for handling the communication, coordination, and data flow across various components.
- **Tenant (Smart_Home)**: The cluster supports multiple tenants, each representing an isolated logical environment. In this project, the tenant is named **Smart_Home**, which encapsulates all the data and operations related to the smart home system.
- **Namespaces**: Under the tenant, the architecture is further divided into **Namespaces**. Each namespace represents a specific domain or category, ensuring better organization and scalability:
  - **Environmental_Monitoring**: This namespace manages topics related to environmental sensors.
  - **Energy_Management**: This namespace is dedicated to topics associated with energy monitoring and control.
  - **Security_Surveillance**: This namespace handles topics related to security and surveillance devices.
- **Topics**: Each namespace contains multiple **Topics**, which are the smallest units of data streams in Pulsar. Topics represent individual sensor devices, allowing fine-grained control over data management. Examples include:
  - **Air_Quality_Monitor** and **Indoor_Climate_Monitoring_Device** under Environmental Monitoring.
  - **Energy_Monitoring_Device** and **Appliance_Control_Device** under Energy Management.
  - **Security_Device** and **Surveillance_Camera** under Security Surveillance.

This architecture ensures:

- **Data Isolation**: Each tenant and namespace is isolated, ensuring data security and separation of concerns.
- **Scalability**: The hierarchical design allows for the seamless addition of new tenants, namespaces, or topics without affecting the existing setup.
- **Efficiency**: By categorizing and organizing topics under specific namespaces, data flow can be managed efficiently, reducing the complexity of operations.

This thoughtful design aligns with the needs of modern distributed systems, making Apache Pulsar a robust platform for managing data streams in smart home environments.

3.2.2 Pulsar Cluster Implementation

3.2.2.1 Initialization

Overview

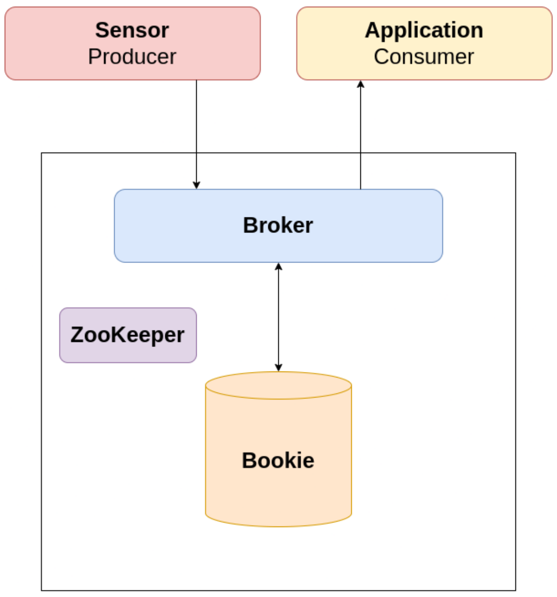


Figure 3.5: System Components

To initialize the Pulsar cluster, we utilized a `docker-compose.yml` file that defines the necessary services and configurations. These services include Zookeeper, Bookkeeper, the Pulsar initialization service, and the Pulsar broker. Each service is configured to ensure smooth communication and proper dependency handling within the Pulsar cluster.

Docker Compose Configuration

The `docker-compose.yml` file organizes the cluster into the following services:

- **Zookeeper**:
  - Zookeeper acts as the metadata manager for the cluster.
  - It is configured with a volume (`./data/zookeeper`) to persist metadata and has health check commands to ensure its readiness.
  - The command initializes the Zookeeper process, applying environment variables to its configuration.
- **Pulsar Initialization Service**:
  - This service initializes the cluster's metadata using the following command:

    ```
    /pulsar/bin/pulsar-admin initialize-cluster-metadata \
        --cluster cluster-a \
        --zookeeper zookeeper:2181 \
        --configuration-store zookeeper:2181 \
        --web-service-url http://broker:8080 \
        --broker-service-url pulsar://broker:6650
    ```
  - Dependencies are defined to ensure Zookeeper is healthy before this service starts.
- **Bookkeeper**:
  - Bookkeeper is responsible for providing persistent storage for Pulsar.
  - It is configured with a volume (`./data/bookkeeper`) for storage and uses environment variables to define the cluster name, metadata service URI, and memory allocations.

    - Dependencies include both Zookeeper and the Pulsar initialization service, ensuring proper sequencing.

- **Broker**:
    - The Pulsar broker manages client connections and data streams.
    - It is configured to expose ports 6650 (for client connections) and 8080 (for administrative tasks).
    - The broker runs a setup script (`setup_pulsar.sh`) after confirming that all required services are running and healthy.

3.2.2.2 Add Tenant

To enhance the organization and efficiency of the Pulsar cluster, we created a tenant to logically group all namespaces and topics. This setup provides better manageability and flexibility within the cluster.
Steps to Add Tenant:

- We first created a tenant named `Smart_Home` to act as a parent group for all related namespaces and topics.
- **Example Command**:

```
/pulsar/bin/pulsar-admin tenants create Smart_Home
```

3.2.2.3 Add Namespaces & Rewrite Namespace Policies

Under the `Smart_Home` tenant, namespaces were created for each major functional domain, with custom retention policies applied to manage data efficiently.
Steps to Add Namespaces and Rewrite Policies:

- **Create Namespaces**:
    - Three namespaces were created to represent the system's major domains:

      * Environmental_Monitoring
      * Energy_Management
      * Security_Surveillance
  - **Example Command**:

```
/pulsar/bin/pulsar-admin namespaces create \
Smart_Home/Environmental_Monitoring
```

    Repeat the command for Energy_Management and Security_Surveillance.

- **Set Namespace Retention Policies**:
  - Retention policies were applied to control the size and time of data retention for each namespace:
      * **Retention Size**: 10M (10 megabytes)
      * **Retention Time**: -1 (infinite retention)
  - **Example Command**:

```
/pulsar/bin/pulsar-admin namespaces set-retention \
Smart_Home/Environmental_Monitoring \
    --size 10M \
    --time -1
```

    Repeat the command for Energy_Management and Security_Surveillance.

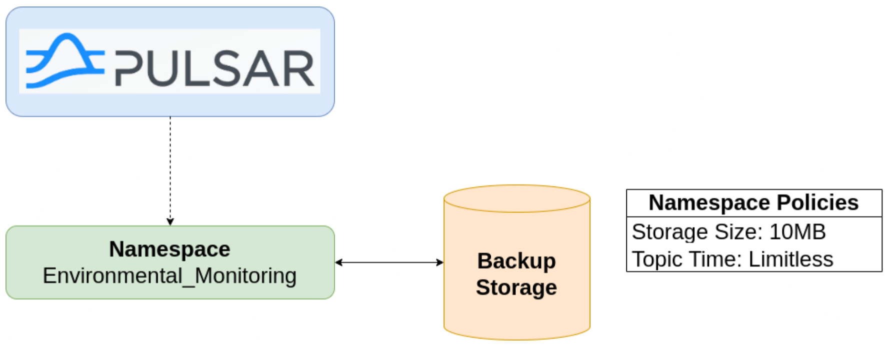


Figure 3.6: Add Namespaces & Change Policies

3.2.2.4 Add Topics

After creating namespaces, topics were added to represent individual data streams for specific devices within each namespace. This granular organization ensures efficient data handling and scalability.

Steps to Add Topics:

- Topics were created for each namespace to represent specific devices. For example:
  - **Environmental Monitoring**:
    - `Air_Quality_Monitor`
    - Example Command:

```
/pulsar/bin/pulsar-admin topics create \
persistent://Smart_Home/Environmental_Monitoring/\
Air_Quality_Monitor
```

Repeat similar commands to create other topics like `Indoor_Climate_Monitoring_Device`, `Energy_Monitoring_Device`, `Appliance_Control_Device`, `Security_Device`, and `Surveillance_Camera`.

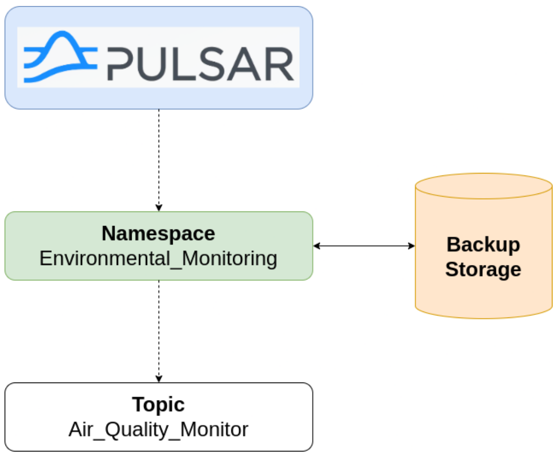


Figure 3.7: Add Topics

3.2.3 Implementing Producer & Consumer

3.2.3.1 Overview

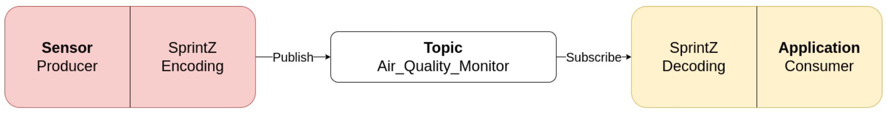


Figure 3.8: Producer and Consumer Workflow

The implementation of the Producer and Consumer follows a structured workflow, as illustrated in Figure 3.8. The goal of this implementation is to facilitate efficient data transmission and processing by leveraging SprintZ compression and Apache Pulsar's topic-based messaging system.

Components of the Workflow:

- **Sensor (Producer):**
  - The producer is responsible for collecting raw data from a sensor (e.g., `Air_Quality_Monitor`).
  - This data is then encoded using the SprintZ compression algorithm to reduce its size before transmission.
- **Topic:**
  - The encoded data is published to an Apache Pulsar topic, such as `persistent://Smart_Home/Environmental_Monitoring/Air_Quality_Monitor`.
  - Topics act as the central data stream hub, enabling efficient data routing to consumers.
- **Application (Consumer):**
  - The consumer subscribes to the topic and retrieves the compressed data.

  - The SprintZ decoding process is then applied to reconstruct the original data.
  - Finally, the application processes the reconstructed data for its intended purpose, such as real-time monitoring or analytics.

This implementation ensures optimized data flow by minimizing bandwidth usage through compression while maintaining the integrity and accessibility of data. The modular approach allows seamless scalability for additional producers, topics, and consumers in the future.

3.2.3.2 Prepare SprintZ Compression

The SprintZ Compression algorithm has already been implemented in our project. However, to enable its integration with the Producer and Consumer programs written in Python, the algorithm must first be compiled into a binary shared object file (`.so`). This compiled file allows the Python program to include SprintZ as an external library, ensuring seamless functionality.

Steps to Compile SprintZ Compression:

1. Clone or pull the project repository from `GitHub` to your local machine.
2. Navigate to the `Demo_CS` directory where the SprintZ implementation resides.
3. Run the following command to build the binary shared object file in-place:

```
cd Demo_CS
python setup.py build_ext --inplace
```

Note: This step is optional as the binary file for SprintZ Compression has already been compiled and included in the project repository. You only need to follow these steps if you wish to recompile the library or make modifications to the SprintZ implementation.

This step ensures that the SprintZ Compression library is fully prepared and functional for its application in the Producer-Consumer pipeline.

3.2.3.3 Implementing Producer & Consumer

The producer-consumer architecture is a crucial component of the system, enabling efficient data flow between sensors and applications. This section describes the workflow and implementation of both the producer and the consumer, which leverage SprintZ encoding and decoding for data compression and reconstruction.

Producer Workflow:

The producer program is responsible for collecting sensor data, compressing it using SprintZ encoding, and publishing it to the designated Pulsar topic. The workflow includes the following steps:

- Dynamically load the pre-compiled SprintZ library (`sprintz_encoder.so`) as an external module in the Python program.
- Read raw sensor data from a CSV file, ensuring compatibility with the expected format.
- Apply SprintZ compression to each row of sensor data. The compression algorithm converts the raw data into an efficient binary format to minimize transmission size.
- Publish the compressed binary data to the Pulsar topic `persistent://Smart_Home/Environmental_Monitoring/Air_Quality_Monitor`.

Consumer Workflow:

The consumer program retrieves compressed messages from the Pulsar topic, decompresses them using SprintZ decoding, and processes the reconstructed data. The workflow includes the following steps:

- Dynamically load the pre-compiled SprintZ library (`sprintz_encoder.so`) for decoding the received messages.
- Subscribe to the Pulsar topic `persistent://Smart_Home/Environmental_Monitoring/Air_Quality_Monitor`.

- Retrieve compressed binary messages from the topic.
- Convert the binary messages back into a readable format using SprintZ decoding. The decoding process uses the previously received data to accurately reconstruct the current data.
- Save the most recently decoded data into a file named `prev_row.txt`. This file is crucial for the decoding process, as SprintZ requires the previous data to properly decode the current message.

Notes:

- The `prev_row.txt` file must exist on the consumer's machine. This file stores the previously decoded data and ensures that SprintZ decompression functions correctly for subsequent messages.
- The data used in this project, including the CSV file, is generated by our group for demonstration purposes. You can find the data in the repository at `GitHub Repository`.
- For more details and the full implementation code, refer to the producer and consumer repository available at `GitHub Repository`.

By implementing this producer-consumer pipeline, the system achieves an efficient and reliable data flow. The use of SprintZ compression reduces the bandwidth required for transmitting data, while the modular design allows for scalability and integration with other components in the Smart Home ecosystem.

Chapter 4

System Deployment & Demonstration

This chapter provides detailed instructions for deploying and demonstrating the Smart Home IoT Environmental and Security Monitoring System on different platforms. The project leverages Apache Pulsar for real-time data streaming and Sprintz compression for efficient IoT data management. These deployment and demonstration steps are essential for replicating the project's functionality on Debian-based OS servers, Debian-based machines, and Windows machines.
Important Note: All the demonstration instructions provided in this chapter can also be found in the `README` file located in the root directory of the project repository on GitHub. For reference, visit `GitHub Repository`.

4.1 Smart Home's Apache Pulsar Deployment on Debian-based OS Server

This guide provides instructions for deploying Apache Pulsar on a Debian-based OS server, a crucial component of the Smart Home IoT Environmental and Security Monitoring System. Apache Pulsar is used for real-time data streaming, complemented by Sprintz compression to manage and process IoT data efficiently for smart home monitoring.

Prerequisites

To deploy Apache Pulsar, ensure the following prerequisites are met:

- Docker and Docker Compose are installed on the server.
- GNOME Terminal or any compatible terminal is available for executing commands.

Initial Setup

1. **Clone the Repository:** Clone the project repository to your server using the following command:

```
git clone https://github.com/longnguyencbct/\
Multidisciplinary_Project_Group4.git
cd Multidisciplinary_Project_Group4/Demo_CE
```

2. **Start the Docker Containers:** Use Docker Compose to build and start the required containers:

```
docker-compose up --build -d
```

   This command builds and initializes the services defined in the `docker-compose.yml` file.

3. **Disable Firewalls:** Ensure that the firewall for port 6650 is disabled to allow the Pulsar Broker to accept producer connections and publish data to consumers.

By following these steps, you will have a fully functional Apache Pulsar deployment ready to handle real-time data streaming for the Smart Home monitoring system.

4.2 Project Demonstration on Debian-based Machines

This section provides a guide for demonstrating the Smart Home IoT Environmental and Security Monitoring System on Debian-based machines. The project uses Apache Pulsar for real-time data streaming and SprintZ compression to manage and process IoT data efficiently.

Prerequisites

Ensure the following tools are installed and available on your machine:

- Docker and Docker Compose
- GNOME Terminal

Setup and Demonstration

1. **Clone the Repository:** Clone the project repository to your local machine using the following commands:

   ```
   git clone https://github.com/longnguyencbct/\
   Multidisciplinary_Project_Group4.git
   cd Multidisciplinary_Project_Group4
   ```

2. **Start the Docker Containers:** Build and start the Docker containers defined in the project:

   ```
   docker-compose up --build -d
   ```

3. **Run Demonstration Scripts:** Execute the following scripts to demonstrate the functionality of producers and consumers for various topics:

   - **Initialize Data Flow:**

     ```
     ./demo/main-demo/demo1_0_init_dataflow
     ```

Starts producer and consumer for the `Air_Quality_Monitor` topic.

- **Producer Initialization:**

```
./demo/main-demo/demo1_1_producer_init_dataflow
```

Runs the producer for the `Air_Quality_Monitor` topic.

- **Consumer Initialization:**

```
./demo/main-demo/demo1_2_consumer_init_dataflow
```

Runs the consumer for the `Air_Quality_Monitor` topic.

- **Multi-topic Data Flow:**

```
./demo/main-demo/demo3_0_multi_dataflow
```

Simulates producers and consumers for multiple topics.

- **Run Multiple Producers:**

```
./demo/main-demo/demo3_1_multi_producers
```

Starts producers for various topics.

- **Run Multiple Consumers:**

```
./demo/main-demo/demo3_2_multi_consumers
```

Starts consumers for various topics.

- **Terminate All Processes:**

```
./demo/main-demo/kill_all_producer_consumer
```

Ensures all producer and consumer processes are stopped.

4. **Manage Containers:** Additional management scripts are available for handling the environment:

- **Recompose Docker Containers:**

```
./docker-recompose.sh
```

Stops, rebuilds, and restarts all containers.

- **Terminate Consumers:**

```
./kill_all_consumer
```

- **Terminate Producers:**

```
./kill_all_producer
```

What Happens in the Process

- **Docker Containers:** The `docker-compose up` command sets up the environment by building and starting containers.
- **Demonstration Scripts:** These scripts simulate real-time data flow by running producers and consumers for specified topics in separate GNOME terminals.
- **Termination Scripts:** Provide functionality to cleanly stop all running processes, ensuring no lingering producers or consumers.

4.3 Project Demonstration on Windows Machines

This section provides a guide for demonstrating the Smart Home IoT Environmental and Security Monitoring System on Windows machines. The project uses Apache Pulsar for real-time data streaming and Sprintz compression to manage and process IoT data efficiently.

Prerequisites

Ensure the following tools are installed and available on your machine:

- Docker and Docker Compose for Windows
- PowerShell

Setup and Demonstration

1. **Clone the Repository:** Clone the project repository to your local machine using the following commands:

```
git clone https://github.com/longnguyencbct/\
Multidisciplinary_Project_Group4.git
cd Multidisciplinary_Project_Group4
```

2. **Start the Docker Containers:** Build and start the Docker containers defined in the project:

```
docker-compose up --build -d
```

3. **Run Demonstration Scripts:** Execute the following scripts to demonstrate the functionality of producers and consumers for various topics:

   - **Initialize Data Flow:**

```
.\demo\main-demo-windows\demo1_0_init_dataflow.ps1
```

     Starts producer and consumer for the `Air_Quality_Monitor` topic.

   - **Producer Initialization:**

```
.\demo\main-demo-windows\demo1_1_producer_init_dataflow.ps1
```

     Runs the producer for the `Air_Quality_Monitor` topic.

   - **Consumer Initialization:**

```
.\demo\main-demo-windows\demo1_2_consumer_init_dataflow.ps1
```

     Runs the consumer for the `Air_Quality_Monitor` topic.

   - **Multi-topic Data Flow:**

```
.\demo\main-demo-windows\demo3_0_multi_dataflow.ps1
```

     Simulates producers and consumers for multiple topics.

   - **Run Multiple Producers:**

```
.\demo\main-demo-windows\demo3_1_multi_producers.ps1
```

Starts producers for various topics.

- **Run Multiple Consumers:**

```
.\demo\main-demo-windows\demo3_2_multi_consumers.ps1
```

Starts consumers for various topics.

- **Terminate All Processes:**

```
.\demo\main-demo-windows\kill_all_producer_consumer.ps1
```

Ensures all producer and consumer processes are stopped.

4. **Manage Containers:** Additional management scripts are available for handling the environment:

- **Recompose Docker Containers:**

```
.\demo\main-demo-windows\docker-recompose.ps1
```

Stops, rebuilds, and restarts all containers.

- **Terminate Consumers:**

```
.\demo\main-demo-windows\kill_all_consumer.ps1
```

- **Terminate Producers:**

```
.\demo\main-demo-windows\kill_all_producer.ps1
```

What Happens in the Process

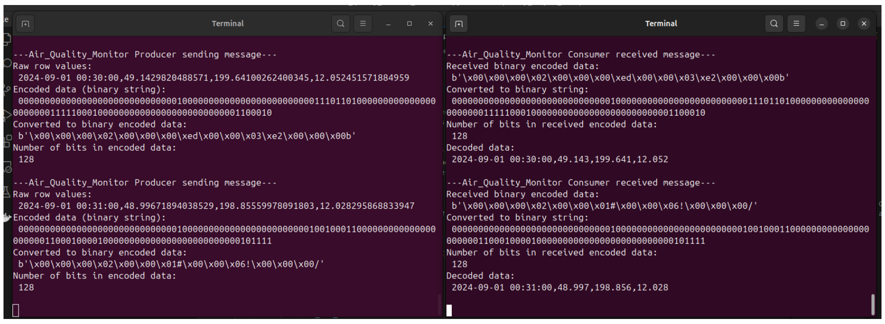
- **Docker Containers:** The `docker-compose up` command sets up the environment by building and starting containers.

- **Demonstration Scripts:** These scripts simulate real-time data flow by running producers and consumers for specified topics in separate PowerShell windows.

- **Termination Scripts:** Provide functionality to cleanly stop all running processes, ensuring no lingering producers or consumers.

4.4 Some Demonstration Images

One Producer, One Consumer

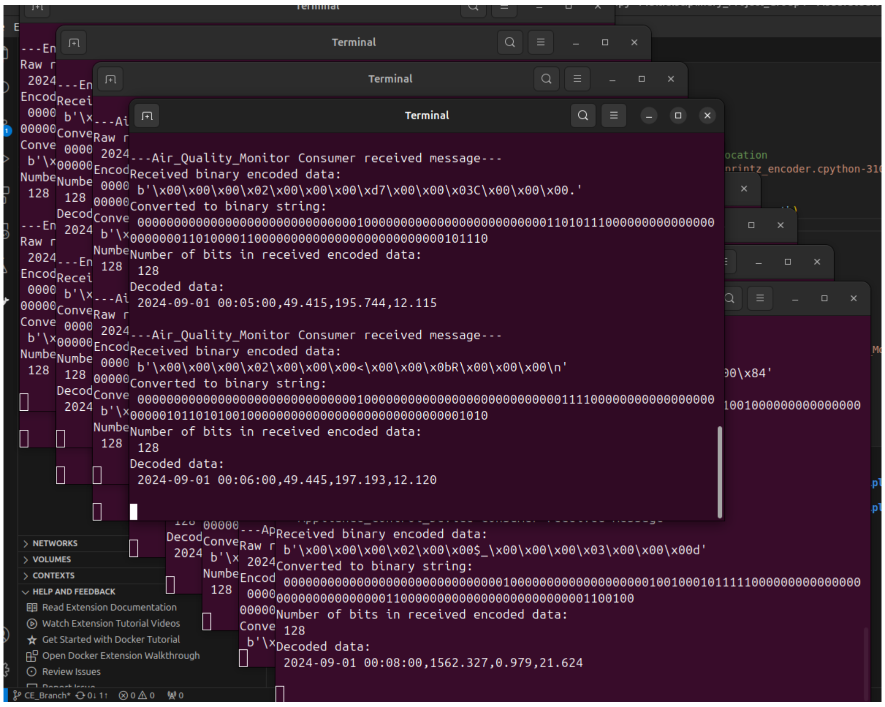


```
---Air_Quality_Monitor Producer sending message---
Raw row values:
 2024-09-01 00:30:00,49.1429820488571,199.64100262400345,12.052451571884959
Encoded data (binary string):
 0000000000000000000000000000001000000000000000000000000011101101000000000000000
0000000111110001000000000000000000000000001100010
Converted to binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x00\xed\x00\x00\x03\xe2\x00\x00\x00b'
Number of bits in encoded data:
 128

---Air_Quality_Monitor Producer sending message---
Raw row values:
 2024-09-01 00:31:00,48.99671894038529,198.85559978091803,12.028295868833947
Encoded data (binary string):
 0000000000000000000000000000001000000000000000000000000100100011000000000000000
0000001100010000100000000000000000000000000101111
Converted to binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x01#\x00\x00\x06!\x00\x00\x00/'
Number of bits in encoded data:
 128
```

```
---Air_Quality_Monitor Consumer received message---
Received binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x00\xed\x00\x00\x03\xe2\x00\x00\x00b'
Converted to binary string:
 0000000000000000000000000000001000000000000000000000000011101101000000000000000
0000000111110001000000000000000000000000001100010
Number of bits in received encoded data:
 128
Decoded data:
 2024-09-01 00:30:00,49.143,199.641,12.052

---Air_Quality_Monitor Consumer received message---
Received binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x01#\x00\x00\x06!\x00\x00\x00/'
Converted to binary string:
 0000000000000000000000000000001000000000000000000000000100100011000000000000000
0000001100010000100000000000000000000000000101111
Number of bits in received encoded data:
 128
Decoded data:
 2024-09-01 00:31:00,48.997,198.856,12.028
```

Figure 4.1: One Producer and One Consumer

Multiple Producers, Multiple Consumers



```
---Air_Quality_Monitor Consumer received message---
Received binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x00\xd7\x00\x00\x03C\x00\x00\x00.'
Converted to binary string:
 0000000000000000000000000000001000000000000000000000000011010111000000000000000
0000000110100001100000000000000000000000000101110
Number of bits in received encoded data:
 128
Decoded data:
 2024-09-01 00:05:00,49.415,195.744,12.115

---Air_Quality_Monitor Consumer received message---
Received binary encoded data:
 b'\x00\x00\x00\x02\x00\x00\x00<\x00\x00\x0bR\x00\x00\x00\n'
Converted to binary string:
 0000000000000000000000000000001000000000000000000000000000111100000000000000000
0000010110101001000000000000000000000000000001010
Number of bits in received encoded data:
 128
Decoded data:
 2024-09-01 00:06:00,49.445,197.193,12.120
```

```
Received binary encoded data:
 b'\x00\x00\x00\x02\x00\x00$_\x00\x00\x00\x03\x00\x00\x00d'
Converted to binary string:
 0000000000000000000000000000001000000000000000000010010001011111000000000000000
0000000000000001100000000000000000000000001100100
Number of bits in received encoded data:
 128
Decoded data:
 2024-09-01 00:08:00,1562.327,0.979,21.624
```

NETWORKS
VOLUMES
CONTEXTS
HELP AND FEEDBACK
Read Extension Documentation
Watch Extension Tutorial Videos
Get Started with Docker Tutorial
Open Docker Extension Walkthrough
Review Issues

Figure 4.2: Multiple Producers and Multiple Consumers

Chapter 5

Conclusion

5.1 Summary

The integration of Apache Pulsar and Sprintz compression in this project successfully addresses key challenges in smart home IoT environments, including data scalability, efficient transmission, and real-time processing. The system's hierarchical structure for managing sensor data enhances scalability and modularity, enabling seamless integration of additional sensors or devices. By leveraging Sprintz's high compression ratios and Pulsar's robust messaging capabilities, the system reduces bandwidth usage while ensuring data integrity and low latency.
Key achievements include:

- **Efficient Data Compression**: Sprintz compression significantly reduces data size without compromising accuracy, optimizing network and storage usage.
- **Scalable Architecture**: Apache Pulsar's tenant-namespace-topic hierarchy ensures modular and scalable data management, suitable for expanding smart home systems.
- **Seamless Integration**: The producer-consumer pipeline demonstrates reli-

able and efficient data flow, enabling real-time monitoring and analytics.

- **Cross-Platform Deployment**: Successful implementation on both Debian-based and Windows systems showcases the system's versatility and adaptability.

This project lays the foundation for future innovations in smart home data processing by proving the efficacy of combining compression algorithms with distributed streaming platforms.

5.2 Future Work

While this project achieves its primary goals, several areas for improvement and expansion remain, providing opportunities for future development:

1. **Enhanced Compression Techniques**:
    - Investigate alternative or hybrid compression algorithms to improve performance for specific data types or environments.
    - Explore adaptive compression methods that dynamically adjust based on the data characteristics.
2. **Real-Time Analytics**:
    - Integrate machine learning models for predictive analytics on sensor data to enhance system intelligence.
    - Develop real-time dashboards for visualizing and managing sensor data streams.
3. **System Optimization**:
    - Optimize the computational efficiency of the producer-consumer pipeline to further reduce latency and processing overhead.

- Improve fault tolerance mechanisms to ensure continuous operation during hardware or software failures.

4. **Expanded Use Cases**:
   - Extend the system to support additional IoT domains, such as healthcare or industrial automation.
   - Incorporate edge computing to process data closer to the source, reducing dependency on centralized systems.
5. **Security and Privacy Enhancements**:
   - Implement advanced encryption mechanisms for secure data transmission.
   - Ensure compliance with data protection regulations, such as GDPR, for user privacy.
6. **Community and Open-Source Collaboration**:
   - Release the project as an open-source platform to foster collaboration and gather community-driven enhancements.
   - Document and publish detailed user guides and tutorials for broader adoption and adaptation.

By addressing these areas, the system can evolve into a comprehensive and versatile platform, driving advancements in smart home and IoT technologies.

References

[1] A. Pulsar, *Multi Tenancy*, https://pulsar.apache.org/docs/4.0.x/concepts-multi-tenancy/tenants, Accessed: 5 December 2024, 2024.

[2] D. BLALOCK, J. GUTTAG, and S. MADDEN, *Sprintz: Time series compression for the internet of things*, Accessed: 10 December 2024, Sep. 2018.

[3] A. Pulsar, *Pulsar Overview*, https://pulsar.apache.org/docs/4.0.x/concepts-overview/, Accessed: 5 December 2024, 2024.

[4] A. Pulsar, *Apache Pulsar*, https://pulsar.apache.org, Accessed: 5 December 2024, 2024.

[5] A. Pulsar, *Concepts Messaging*, https://pulsar.apache.org/docs/4.0.x/concepts-messaging/, Accessed: 5 December 2024, 2024.

[6] G. Polyzos, *Event Streaming in Apache Pulsar*, https://rockthejvm.com/articles/event-streaming-in-apache-pulsar-with-scala/, Accessed: 5 December 2024, 2021.

[7] M. G. M. Needham, *OSA Con 2022 - Building a Real-time Analytics Application with Apache Pulsar and Apache Pinot*, Accessed: 5 December 2024, 2022.

[8] A. Pulsar, *ZooKeeper: Because Coordinating Distributed Systems is a Zoo*, https://zookeeper.apache.org/doc/current/zookeeperOver.html, Accessed: 5 December 2024, 2024.

[9] L. Q. J. Chen, *Apache Pulsar in Practice: vivo's Road to a Cloud-native Messaging and Streaming Experience*, https://rockthejvm.com/articles/event-streaming-in-apache-pulsar-with-scala/, Accessed: 5 December 2024, 2022.