MULTIDISCIPLINARY PROJECT

SMART HOME

GROUP 04

- [CE] NGUYỄN VĂN ĐỨC LONG - 2153535
- [CE] TRẦN NGUYỄN GIA HUY - 2153395
- [CS] PHẠM LÊ QUÂN - 2153749
- [CS] TRƯƠNG HUỲNH ANH DŨNG - 2053622

CONTENT

01 INTRODUCTION

02 SYSTEM DESIGN & TECHNOLOGY

03 SYSTEM IMPLEMENTATION

04 DEMONSTRATION

INTRODUCTION

MOTIVATION

Image source: https://ru.pinterest.com/pin/114701121748458079/

The rapid adoption of smart home systems has led to a significant **increase** in **data generated** by sensors for air quality, energy consumption, and security,... Managing this data poses challenges in **scalability** and **efficiency**, especially with bandwidth and storage limitations on servers.

Apache Pulsar offers a robust solution as a distributed streaming platform capable of handling **high-throughput**, **low-latency** data streams. By integrating **SprintZ** compression, the project aims to optimize data transmission by **reducing data size**, ensuring **efficient bandwidth** usage, and **improving** real-time data **delivery**.

GOAL

Image source: https://www.freepik.com/icon/smart-house_1384368

This project aims to address the challenges of data efficiency, scalability, and performance in smart home systems by focusing on the following goals:

- **Improving Transmission Efficiency**: Reducing data size with SprintZ compression to lower bandwidth usage.
- **Enhancing Throughput:** Streamlining data flow to handle more data with fewer resources.
- **Ensuring Seamless Integration**: Supporting real-time compatibility with existing devices.
- **Achieving Scalability and Reliability**: Scaling effortlessly while maintaining robust data handling.

SYSTEM DESIGN & TECHNOLOGY

SYSTEM DESIGN & TECHNOLOGY

Overall

Compression Technology:
SprintZ

Data Streaming Technology:
Apache Pulsar

SYSTEM DESIGN & TECHNOLOGY OVERALL

Overall - System Design

SYSTEM DESIGN & TECHNOLOGY OVERALL

Overall - 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
- Energy Management
- Security Surveillance

SYSTEM DESIGN & TECHNOLOGY OVERALL

Overall - Data Compression

SYSTEM DESIGN & TECHNOLOGY OVERALL

Overall - Data Compression

- **Producer Side**: Compresses data to reduce size, lowering bandwidth usage and improving throughput.
- **Broker Transmission**: Smaller data sizes enable efficient handling of more messages, reducing latency and preventing bottlenecks.
- **Consumer Side**: Decompression requires resources, which can lower throughput if not optimized.

Key Benefits:

- Enhances **scalability** and **efficiency**.
- **Reduces** network and storage **load**.
- **Balances size-speed** trade-off for optimal performance.

COMPRESSION TECHNOLOGY: SPRINTZ

Problem

- Sensor-generated time-series data is becoming one of the largest contributors to global data streams, driven by IoT, smart devices and monitoring systems.
- Devices generating this data, such as IoT sensors and embedded systems, have strict limitations on memory, computational power and energy consumption.
- Transmission Challenges:
  - Wireless transmission consumes significant power, making it infeasible to transmit uncompressed data continuously.
  - Limited resources make traditional compression algorithms unsuitable due to high memory and latency demands.

INTRODUCTION

Solution

- Tailored for Time-Series Data:
  - Specially designed for data with temporal correlations, low bit-widths and high sampling rates.
  - Capable of handling multivariate datasets with minimal preprocessing.
- High Performance and Efficiency:
  - Decompression speeds exceeding 3 GB/s on a single thread.
  - Lossless compression ensures data integrity, maintaining every detail for accurate analysis.
- Efficient Resource Usage:
  - Operates with less than 1 KB of memory, making it ideal for resource-constrained environments.
  - Processes data in small blocks, minimizing latency and supporting real-time applications.
- Leverages advanced methods like vectorized forecasting, bit-packing, run-length encoding and entropy coding.

INTRODUCTION

Solution

More specifically, existing methods (e.g., [19, 20, 25, 27, 30, 41, 45, 48, 51]) violate one or more of the following design requirements:

(1) **Small block size**. On devices with only a few kilobytes of memory, it is not possible to buffer large amounts of data before compressing it. Moreover, even with more memory, buffering can add unacceptable latency; for example, a smartwatch transmitting nine axes of 8-bit motion data at 20Hz to a smartphone would need to wait $10000/(9 \times 1 \times 20) = 56$ seconds to fill even a 10KB buffer. This precludes using this data for gesture recognition and would add tremendous user interface latency for step counting, activity recognition, or most other purposes.

(2) **High decompression speed**. While the device collecting the data may not need to decompress it, it is desirable to have an algorithm that could also function well in a central database. This eliminates the need to transcode the data at the server and simplifies the application. In a database, time series workloads are not only read-heavy [6, 10, 14], but often necessitate materializing data (or downsampled versions thereof) for visualization, clustering, computing correlations, or other operations [14]. At the same time, writing is often append-only [14, 55]. As a result, decompression speed is paramount, while compression speed need only be fast enough to keep up with the rate of data ingestion.

(3) **Lossless**. Given that time series are almost always noisy and often oversampled, it might not seem necessary to compress them losslessly. However, noise and oversampling 1) tend to vary across applications, and 2) can be addressed in an application-specific way as a preprocessing step. Consequently, instead of assuming that some level of downsampling or some particular smoothing will be appropriate for all data, it is better for the compression algorithm to preserve what it is given and leave preprocessing up to the application developer.

The primary contribution of this work is Sprintz, a compression algorithm for time series that offers state-of-the-art compression ratios and speed while also satisfying all of the above requirements. It requires <1KB of

Source: https://arxiv.org/pdf/1808.02515.pdf

KEY FEATURES

What Makes Sprintz Unique?

1. High Efficiency	*2. Minimal Resource Footprint*
• Fast compression and decompression. • Decompression speeds: >3 GB/s on a single thread.	• Operates with <1 KB of memory. • Processes small data blocks (8 samples), reducing latency.

KEY FEATURES

What Makes Sprintz Unique?

3. Lossless Compression	*4. Flexible Forecasting*
• Preserves data integrity, ensuring no information is lost. • Enables accurate downstream analysis for critical applications.	• Delta Coding: Predicts current samples using prior samples. • FIRE (Fast Integer Regression): Adapts predictions in real time with an online learning algorithm.

KEY FEATURES

What Makes Sprintz Unique?

5. Efficient Techniques	*6. Scalability*
• Bit-packing: Uses minimal bits for storage. • Run-length Encoding: Compresses repetitive data efficiently. • Entropy Coding (e.g., Huffman): Reduces data size further.	• Supports multivariate time-series data. • Efficiently handles growing datasets with multiple dimensions.

DATA STREAMING TECHNOLOGY: APACHE PULSAR

Introduction

Messaging Concepts

Key Components

INTRODUCTION

PULSAR

Definition

- Apache Pulsar is a distributed messaging and streaming platform initially developed by **Yahoo**, now managed by the **Apache Software Foundation**.
- It provides highly scalable, low-latency data transfer.

Importance

- Pulsar is built on the **publish-subscribe pattern** (often abbreviated to pub-sub).
- Producers **publish messages** to topics; Consumers **subscribe to those topics**, **process** incoming messages, and send **acknowledgments** to the broker when processing is finished.

Source: https://pulsar.apache.org/docs/4.0.x/concepts-messaging/

KEY FEATURES

PULSAR

Geo-replication

Low latency

Scalability

Client API for Java, Go, Python and C++

Subscription types

Persistent storage

- **Pulsar's combination of messaging and streaming supports real-time communication**
- **Ideal for use cases such as IoT, financial services, and real-time analytics.**

MESSAGING CONCEPTS

Tenants

- A **tenant** is the highest-level entity, representing an **isolated group** within a Pulsar instance.
- It **enables multiple organizations** or teams to **share a Pulsar** instance while keeping data separate.

MESSAGING CONCEPTS

Namespaces

- A **namespace** is a logical grouping 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.

MESSAGING CONCEPTS

Topics

- BookKeeper ensures **scalability** and **fault tolerance** by **evenly distributing segments** across bookies - called **Topics**.
- This allows Pulsar topics to **scale** beyond the capacity of a single node.

SUBSCRIPTION TYPES

- **Exclusive**: only one consumer is allowed to read messages using the subscription at any point in time.
- **Failover**: supports multiple consumers, but only one is active at a time. If the active consumer fails, the next consumer in line takes over.
- **Shared**: allows multiple consumers to attach to the same subscription. Messages are distributed among all active consumers with no guaranteed order.
- **Key-Shared**: similar to Shared but maintains order for messages with the same key. Ensures that messages with the same key are sent to the same consumer, preserving order for specific keys.

Source: https://rockthejvm.com/articles/event-streaming-in-apache-pulsar-with-scala

KEY COMPONENTS OF APACHE PULSAR

BROKER

BOOKKEEPER

ZOOKEEPER

BROKER

Source: A Scalable Data Pipeline for Realtime Geofencing Using Apache Pulsar, K. Sundar Rajan, A. Vishal & Chitra Babu

Definition

Brokers form the stateless serving layer in Apache Pulsar. They are **responsible** for **handling client connections**, **managing routing**, and **distributing messages** to topics. Brokers do not store messages locally because data is stored in **BookKeeper**

How Does It Work

Each **topic** partition is assigned to a **broker.** Producers and consumers connect to the owner broker via topics for message production and consumption. In case of **broker failure**, **topic ownership** is **transferred** to **another broker** without moving data.

Use Case

Brokers are essential for **managing client connections** and **routing messages** between producers and consumers, providing **scalability** and **high availability** in distributed messaging environments.

BOOKKEEPER

Source: https://streamnative.io/blog/guide-apache-pulsar-compare-features-architecture-to-apache-kafka

Definition

BookKeeper is the persistence layer for Apache Pulsar, responsible for **storing messages** across **multiple bookies** in a cluster.

How Does It Work

BookKeeper **stores topic partitions** as distributed **logs**. Each **log** is divided into **segments**, which are **replicated across bookies** to ensure **redundancy**. New segments are created based on time or size limits, or ownership changes.

Use case

BookKeeper ensures **scalability** and **fault tolerance** by **evenly distributing segments** across bookies. This allows Pulsar topics to scale beyond the capacity of a single node.

ZOOKEEPER

Source: https://www.waitingforcode.com/apache-pulsar/apache-pulsar-global-architecture-local-setup/read

Definition

ZooKeeper is a centralized service that **manages metadata** and **coordinates** the distributed **components** within an Apache Pulsar cluster.

How Does It Work

ZooKeeper stores metadata like **broker locations**, **topic assignments**, and **ledger information**. It also ensures **synchronization** by facilitating **leader election** and managing state consistency across brokers and bookies.

Use case

Essential for maintaining **cluster stability**, ZooKeeper enables high availability by **coordinating failover** and **balancing workloads**, supporting Pulsar's scalability and fault tolerance.

SYSTEM IMPLEMENTATION

SYSTEM IMPLEMENTATION

SprintZ

Apache Pulsar

SYSTEM IMPLEMENTATION: SPRINTZ

Main Components

- 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.
  - Compressed data structure: Each data block is organized so that it is easy to retrieve and decompress, ensuring fast processing speed.
- Run-Length Encoding:
  - Record the number of zero value
  - Achieve extremely high compression ratios in cases where the data is stable
- Entropy Coding:
  - After compressing with bit packing and RLE, Sprintz compresses data based on the frequency of occurrence

We also assess three variations of SPRINTZ, corresponding to different speed/ratio tradeoffs:

(1) **SprintzFIRE+Huf**. The full algorithm described in Section 4.
(2) **SprintzFIRE**. Like SprintzFIRE+Huf, but without Huffman coding.
(3) **SprintzDelta**. Like SprintzFIRE, but with delta coding instead of FIRE as the forecaster.

Source: https://arxiv.org/pdf/1808.02515.pdf

SYSTEM IMPLEMENTATION: SPRINTZ

air_quality_monitors.bin 136 KB

air_quality_monitors.csv 634 KB

SYSTEM IMPLEMENTATION: SPRINTZ

SYSTEM IMPLEMENTATION: APACHE PULSAR

Overall Pulsar Architecture

SYSTEM IMPLEMENTATION: APACHE PULSAR

Overall Pulsar Architecture

Pulsar Cluster: At the top level of the architecture is the Pulsar Cluster, which acts as the central data management system.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Overall Pulsar Architecture

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.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Overall Pulsar Architecture

Namespaces: Under the tenant, the architecture is further divided into **Namespaces**. Each namespace represents a specific domain or category, ensuring better organization and scalability.

In our project, there are 3 Namespaces:

- Environmental_Monitoring
- Energy_Management
- Security Surveillance

SYSTEM IMPLEMENTATION: APACHE PULSAR

Overall Pulsar Architecture

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.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Pulsar Cluster Implementation

To initialize the Pulsar cluster, we utilized a **docker-compose.yml** file that defines the necessary services and configurations.

Note: We have set up everything in docker-compose.yml file.

Our .yml file is inspired (but we made some adjustments) by the guidance of the Official Apache Pulsar website. You can access the guide here:

https://pulsar.apache.org/docs/next/getting-started-docker-compose/

SYSTEM IMPLEMENTATION: APACHE PULSAR

Pulsar Cluster Implementation

- **Zookeeper**: Manages cluster metadata, persists data using a volume, and ensures readiness through health checks.
- **Pulsar Initialization Service**: Sets up the cluster's metadata, ensuring Zookeeper is healthy before starting.
- **Bookkeeper**: Provides persistent storage for Pulsar, using a volume for data and environment variables for configuration. It starts after **Zookeeper** and the **Pulsar Initialization Service**.
- **Broker**: Manages client connections and data streams, exposing ports 6650 (clients) and 8080 (admin). It runs a setup script once **all dependencies** are confirmed healthy.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Add Tenant

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

```
docker exec -it broker /bin/bash
/pulsar/bin/pulsar-admin tenants create Smart_Home
```

SYSTEM IMPLEMENTATION: APACHE PULSAR

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.

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

SYSTEM IMPLEMENTATION: APACHE PULSAR

Add Namespaces & Rewrite Namespace Policies

Retention policies were applied to control the size and time of data retention for each namespace

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

SYSTEM IMPLEMENTATION: APACHE PULSAR

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

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

SYSTEM IMPLEMENTATION: APACHE PULSAR

Revise

Repeating the presented commands for other namespaces & topics, we will have this architecture. **OR...**

Type these commands to set up The Apache Pulsar server for **Smart_Home** automatically:

```
git clone https://github.com/longnguyencbct/Multidisciplinary_Project_Group4.git
cd Multidisciplinary_Project_Group4/Demo_CE
docker-compose up --build -d
```

SYSTEM IMPLEMENTATION: APACHE PULSAR

Revise

You can also read the guidance on how to set up Apache Pulsar Cluster on your server on our Github Repository: https://github.com/longnguyencbct/Multidisciplinary_Project_Group4/blob/main/Demo_CE/README.md

SYSTEM IMPLEMENTATION: APACHE PULSAR

Implementing Producer & Consumer

This implementation ensures optimized data flow by minimizing bandwidth usage through compression while maintaining the integrity and accessibility of data.

SYSTEM IMPLEMENTATION: APACHE PULSAR

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.

```
git clone https://github.com/longnguyencbct/Multidisciplinary_Project_Group4.git
cd Multidisciplinary_Project_Group4/Demo_CS
python setup.py build_ext -- inplace
```

SYSTEM IMPLEMENTATION: APACHE PULSAR

Implementing Producer

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

- 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.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Implementing Consumer

The consumer program retrieves compressed messages from the Pulsar topic, decompresses them using SprintZ decoding, and processes the reconstructed data

- Dynamically load the pre-compiled SprintZ library (sprintz_encoder.so) for decoding the received messages.
- Subscribe to the Pulsar topic.
- 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.

SYSTEM IMPLEMENTATION: APACHE PULSAR

Implementing Producer & Consumer

For more details and the full implementation code, refer to the producer and consumer repository available at https://github.com/longnguyencbct/Multidisciplinary_Project_Group4/tree/main/Demo_CE/Topics/Air_Quality_Monitor

We have also prepared a docker-compose.yml file to set up a demonstration environment where all producers and consumers for every topic are configured. More details about it are forthcoming.

DEMONSTRATION

DEMONSTRATION

Demo 1: Normal dataflow of 1 topic

Demo 2: Consumer temporary down

Demo 3: Dataflows on many topics

DEMO 1: NORMAL DATAFLOW OF 1 TOPIC

Computer 1: Producer

```
cd demo/main-demo
or
cd demo/main-demo-windows
```

```
./docker_recompose
```

```
./demo1_1_producer_init_dataflow
```

Computer 2: Consumer

```
cd demo/main-demo
or
cd demo/main-demo-windows
```

```
./docker_recompose
```

```
./demo1_2_consumer_init_dataflow
```

DEMO 1+2: CONSUMER TEMPORARY DOWN

Computer 1: Producer

Computer 2: Consumer

```
./kill_all_consumer
```

```
./demo2_1_consumer_rerun
```

DEMO 3: DATAFLOWS ON MANY TOPICS

Computer 1: Producer

```
./kill_all_producer_consumer
```

```
./demo3_1_multi_producers
```

Computer 2: Consumer

```
./kill_all_producer_consumer
```

```
./demo3_2_multi_consumers
```

DEMO 3+2: CONSUMER TEMPORARY DOWN

Computer 1: Producer

Computer 2: Consumer

```
./kill_all_consumer
```

```
./demo2_2_multi_consumers_rerun
```

THANK YOU FOR LISTENING