MIDTERM REPORT PROJECT

UNIVERSITÀ DI BOLOGNA
DISTRIBUTED SOFTWARE SYSTEMS
APACHE FLINK MIDDLEWARE

Author: Carlos Lozano Alemañy

Contents

Chapter 1

Description of a Middleware Framework

1.1 Definiton

Middleware is a crucial component in the landscape of distributed systems, acting as an intermediary layer between different software applications. It provides a set of services and functionalities that facilitate communication, integration, and interaction among diverse components in a distributed environment. Essentially, middleware serves as the "glue" that enables seamless collaboration between applications running on different machines or nodes.

1.2 Funcionality

The primary goal of employing middleware frameworks is to abstract the complexities of distributed computing, offering a standardized way for software components to communicate and exchange data. By providing a common set of services, middleware simplifies the development and maintenance of distributed systems. This abstraction allows developers to focus on the business logic of their applications without getting entangled in the intricacies of handling communication protocols, data serialization, and network management.

1.3 Achieved key goals

1.3.1 Interoperability

Middleware enables applications developed in different languages or running on different platforms to communicate seamlessly. This promotes interoperability, allowing for the integration of diverse technologies within a distributed system.

1.3.2 Scalability

As distributed systems grow in complexity and scale, middleware provides mechanisms for handling increased workloads and data traffic. It supports the scalable expansion of applications across multiple nodes or servers.

1.3.3 Reliability and Fault Tolerance

Middleware often includes features that enhance the reliability of distributed systems. This includes mechanisms for error handling, fault tolerance, and recovery, ensuring the robustness of the overall system. Abstraction of Network Details: Middleware shields developers from the low-level details of network communication. It abstracts the complexities of managing connections, data serialization, and synchronization, allowing developers to focus on building functional features.

Chapter 2

Context

2.1 Introduction of Apache Flink

Apache is a powerful and versatile open-source framework designed for distributed stream and batch data processing. It emerged as a prominent solution for real-time analytics, offering capabilities for handling massive datasets with low-latency processing. In fact the name Flink, (it means fast or agile in German), was selected to honor the style of this stream and batch processor.

2.2 Main use cases

The use cases are the diverse scenarios in which the framework excels, showcasing its flexibility and effectiveness. The main use cases of Apache Flink are the following ones:

2.2.1 Event-driven Applications

An event-driven application is a type of application that maintains a state and processes events from one or more event streams. It responds to incoming events by initiating computations, updating its state, or triggering external actions. Event-driven applications represent a progression from conventional application designs that segregate computing and data storage functions. In this architectural approach, these applications both read data from and store data in a distant transactional database. Unlike traditional setups, event-driven applications rely on stateful stream processing. In this model, data and computation occur in close proximity, facilitating local data access through in-memory or disk storage. To ensure fault tolerance, the application periodically creates checkpoints that are stored in a remote persistent storage system.

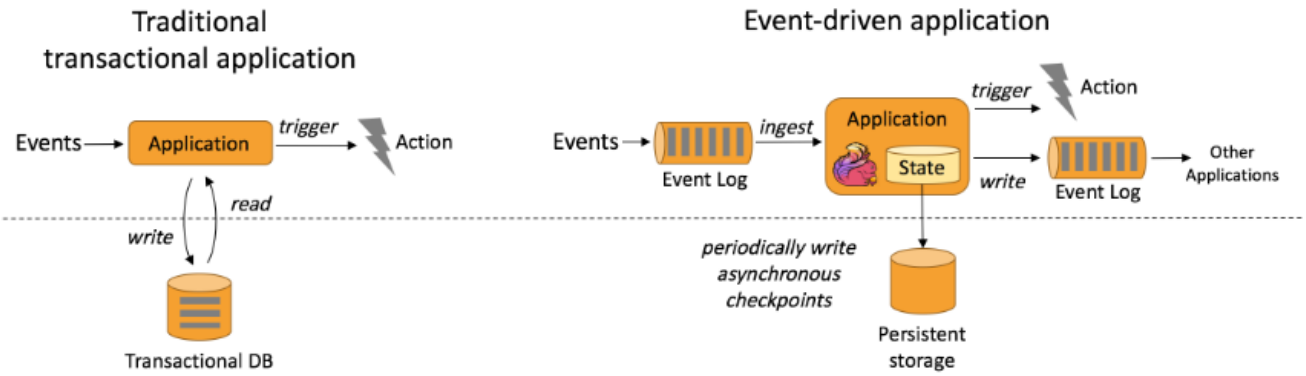

Figure 2.1: Difference between the traditional application architecture and event-driven applications.

Example of event-driven application

Alibaba Group, a substantial ecommerce entity, engages in transactions between buyers and suppliers through its web portal. The company relies on a customized version of Flink, known as Blink, to generate online recommendations. Leveraging a genuine streaming engine like Flink offers a distinctive advantage— it allows real-time consideration of purchases made throughout the day when suggesting products to users. This becomes especially crucial during peak activities on special occasions or holidays. This scenario exemplifies a use case where the efficiency of stream processing holds a significant edge over traditional batch processing.

2.2.2 Batch processing

Apache Flink offers support for batch processing, a feature particularly advantageous for tasks involving the simultaneous handling of extensive datasets. This capability proves crucial in scenarios like data storage, where the efficiency of managing massive volumes of data is of utmost importance. Analytical tasks focus on extracting valuable information and insights from raw data. Traditionally, analytics involve the execution of batch queries or applications on limited datasets containing recorded events. To incorporate the latest data into the analysis results, it must be appended to the existing dataset, and subsequently, the query or application is rerun. The outcomes are then stored in a dedicated storage system or presented as reports.

2.2.3 Real time data processing (Stream Processing)

Utilizing an advanced stream processing engine, analytics can be conducted in real-time. Instead of working with fixed datasets, streaming queries or applications process real-time event streams continuously, generating and updating results as events are received. These results can be stored either in an external database or as an internal state. Dashboard applications have the capability to retrieve the most recent results either from the external database or by directly querying the internal state of the application.

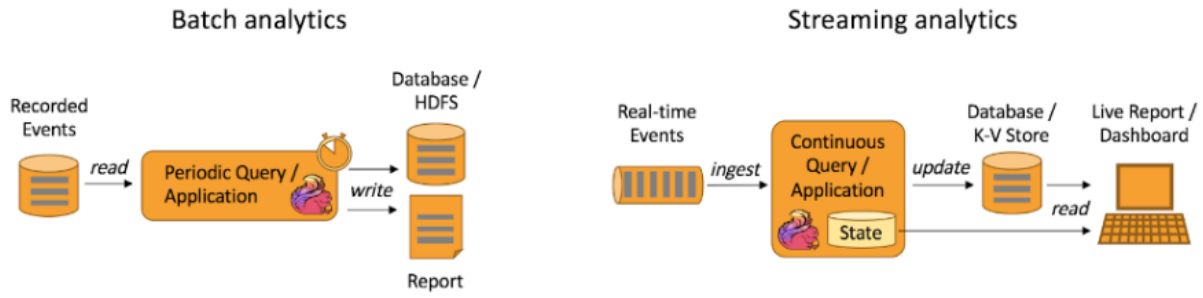

Figure 2.2: Apache Flink supports streaming as well as batch analytical applications.

2.3 Possible Scenario

There is a company that offers an online music streaming service, with millions of users worldwide. Users interact with the platform in a variety of ways: listen to songs, create playlists, rate songs, skip songs, etc.

2.3.1 Challenges

Time Personalization

Users expect a personalized experience. They want song recommendations based on their current tastes and activities.

Trend Detection

The platform must be able to detect real-time trends and adjust popular playlists or recommendations accordingly.

User Behavior Monitoring

It is crucial to monitor user behavior to understand which features are most used, which genres are popular at certain times, and how users interact with the platform.

Live Updates

Live updates, such as the addition of new songs, changes in user ratings, and real-time events, should be quickly reflected in recommendations and playlists.

2.3.2 Using Apache Flink

Real-Time Processing

Apache Flink can process user interactions in real-time, capturing events such as plays, ratings, and playlist creation as they occur.

Personalized Recommendations

Using Flink's real-time processing capability, personalized recommendations can be generated for users based on their listening history and current activities.

Trend Detection

Flink analyzes emerging patterns in streaming data to detect trends and dynamically adjust playlists and popular recommendations.

Real-Time Monitoring

Flink enables real-time monitoring of user behavior, providing instant analytics on how users interact with the platform.

Integration with External Data Sources

Flink can integrate with external data sources, such as social networks, to enrich analytics and recommendations with additional information on preferences and trends.

2.3.3 Expected Results

In this scenario, Apache Flink becomes an essential tool for processing and analyzing large volumes of data efficiently and in real-time, enabling a highly personalized and dynamic music streaming experience and also actionable analytics.

Chapter 3

Main architectural drivers

In the framework context, architectural drivers represent the core considerations that shape design choices and functionality. They are high-level goals guiding development to meet specific non-functional requirements. For instance, Apache Flink's architectural drivers are the ones below.

3.1 Scalability

The framework achieves scalability through horizontal scaling, allowing the addition of more resources to handle increasing workloads. This is particularly advantageous in scenarios where data volumes grow rapidly, ensuring that the system can adapt to changing demands.

3.2 Fault Tolerance

In distributed environments, the likelihood of individual components failing is higher. Apache Flink addresses this challenge by implementing robust fault tolerance mechanisms like using checkpoints by keeping track of intermediate conditions of state such that the system can be accurately reset if there is a failure.

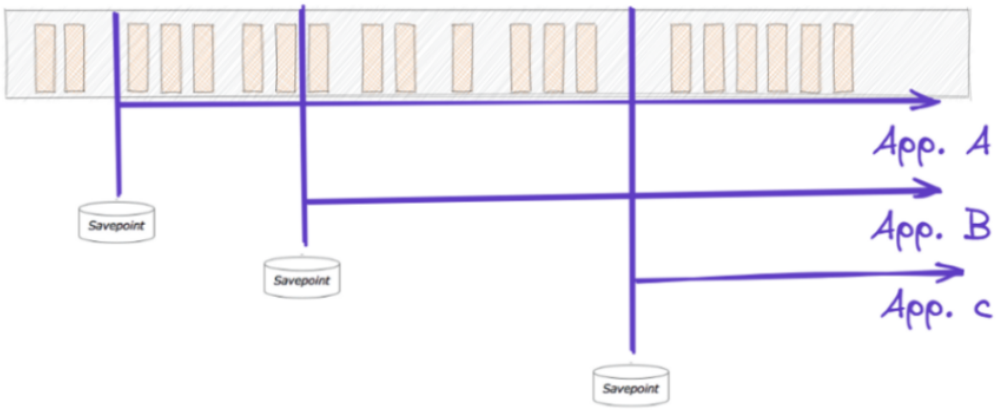

Figure 3.1: Intermediate conditions of state

3.3 Low Latency

Apache Flink achieves low-latency processing through a combination of design principles, optimizations, and features that prioritize minimizing the time it takes to process and respond to events. Here are key factors contributing to Flink's low-latency capabilities:

3.3.1 Pipelined Data Processing

Flink uses a pipelined execution model, where the data processing pipeline is broken into stages, and each stage processes data as soon as it becomes available. This reduces the overall processing time as data flows through the system without waiting for the entire pipeline to complete.

3.3.2 Event Time Processing

Flink supports event time processing, allowing the system to process events based on the time they occurred rather than when they arrive at the system. This feature is crucial for scenarios where events may arrive out of order, ensuring accurate and low-latency processing.

Chapter 4

Structure

Apache Flink's architecture follows a master/worker pattern, where a central coordinator (master) manages and distributes tasks to worker nodes. There are two distinct processes in Flink, illustrated in the following diagram:

4.1 Job Managers (Master)

JobManagers are tasked with distributing tasks among TaskManagers, overseeing checkpoint management, and handling recovery in the event of failures. To ensure high availability, it is advisable to configure multiple JobManagers where one operates as the leader and the others as standbys. The leader and standby JobManagers remain synchronized, and in the event of leader failure, a standby JobManager seamlessly assumes the leader role.

4.2 Task Managers (Workers)

TaskManagers play a crucial role in executing tasks and managing buffers for operators, among other responsibilities. Each TaskManager operates as a Java Virtual Machine (JVM) process, and tasks are executed within multiple threads known as task slots. TaskManagers can be equipped with multiple slots, with one slot potentially shared by multiple subtasks within a job. It's important to note that the client is not involved in program execution; its sole purpose is to submit the job, and it may remain connected solely to retrieve the job's status at regular intervals.

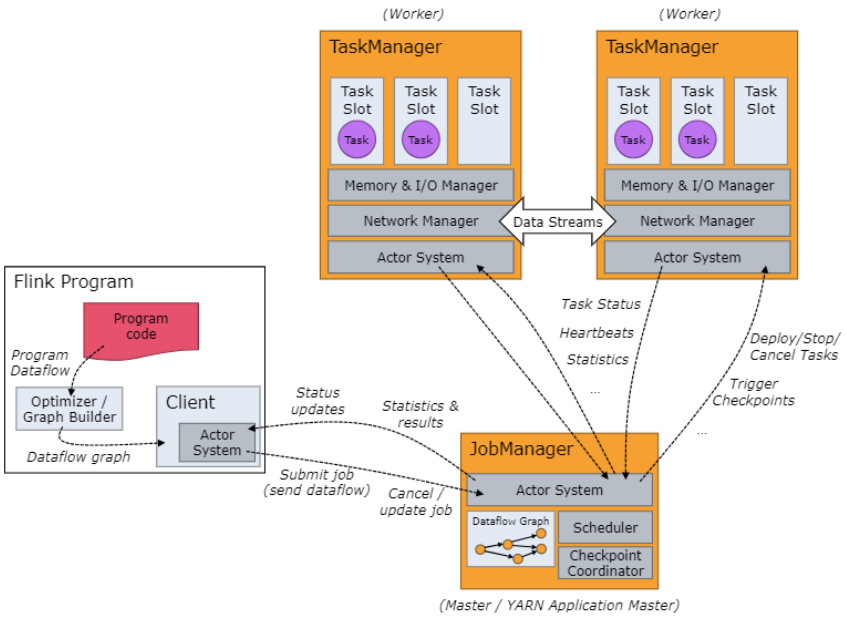

Figure 4.1: Flink Architecture

Chapter 5

Behavior

Apache Flink's behavior is characterized by its ability to handle both stream processing and batch processing in a single system.

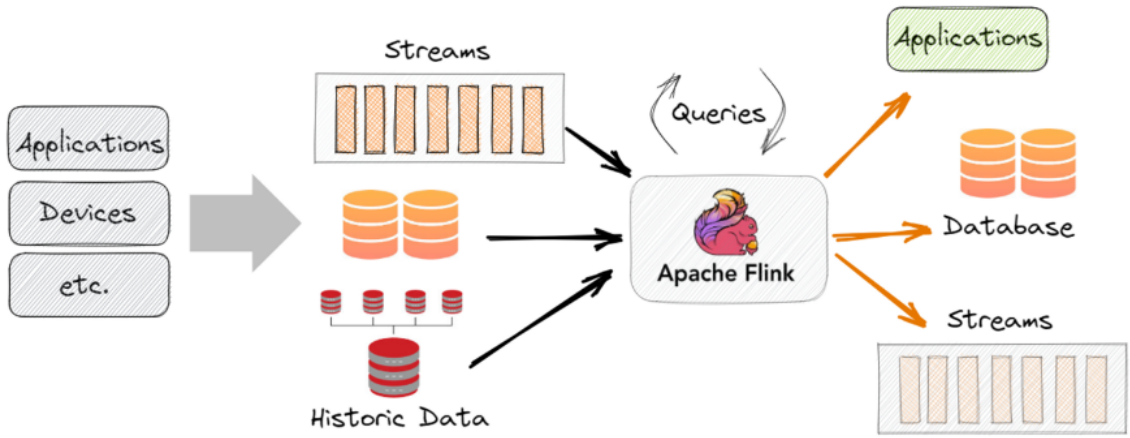

Figure 5.1: Flink Behavior

As we can see in the image, the behavior of the Apache Flink framework could be described in 3 parts:

5.1 Part 1: Data Ingest

In the first phase, Apache Flink receives data from various sources, be it streaming, batch or databases. This is accomplished through built-in connectors that allow interaction with various storage systems and data sources.

5.2 Part 2: Data Processing

In the processing stage, Flink builds a pipeline that describes how to transform and analyze the data. Flink operators will be applied to the data streams. In addition, the system manages state, processes events based on their time of occurrence, and performs checkpoints to ensure fault tolerance.

5.3 Part 3: Delivering Results

Once processed, the data is delivered to various destinations. Flink has connectors that allow integration with external systems, making it possible to write processed data to file systems, databases, messaging systems or external applications.

Chapter 6

Rationale

Apache Flink has the form it does due to strategic design choices aimed at addressing specific challenges and requirements within the realm of distributed data processing. The framework's architecture and features are shaped by key considerations to ensure its effectiveness in various use cases. Here's why Apache Flink has the form it does:

Unified Real-Time and Batch Processing

Flink is designed to provide a unified platform for both real-time and batch processing. This choice acknowledges the need for a versatile framework that can seamlessly handle diverse data processing scenarios.

Exactly-Once Semantics and Checkpoints

The emphasis on exactly-once semantics and robust checkpointing mechanisms is crucial for ensuring data integrity and fault tolerance in distributed systems, a critical requirement for many applications.

Event Time Processing for Accuracy

Support for event time processing acknowledges the complexities of real-world data, allowing Flink to process events based on their timestamps rather than the order of arrival, ensuring accurate and meaningful analysis.

Pipelined Data Processing for Low Latency

Flink's pipelined execution model contributes to low-latency processing by breaking down the data processing pipeline into stages. This design minimizes processing times and allows Flink to respond quickly to incoming data.

Dynamic Scaling and Horizontal Scalability

The support for dynamic scaling and horizontal scalability enables Flink to efficiently handle varying workloads by adding or removing resources during runtime. This design ensures optimal resource utilization.

Stateful Processing for Complex Applications

Flink's emphasis on stateful processing addresses the needs of applications requiring the maintenance and updating of state across multiple events. This capability is essential for building complex, event-driven applications.

Rich Set of Connectors for Ecosystem Integration

Flink's provision of a diverse set of connectors facilitates seamless integration with various data sources and sinks. This design promotes interoperability within the broader data processing ecosystem.

Adaptive Query Optimization for Performance

Flink's adaptive query optimization recognizes the dynamic nature of data and adapts its execution plan accordingly. This design optimizes performance based on evolving conditions, enhancing efficiency.

Community and Ecosystem Collaboration

Active community engagement and integration with popular big data ecosystems are essential for ongoing development, fostering collaboration, and ensuring compatibility with a broad range of tools and systems.

Windowing Operations for Time-Based Analysis

Flink's support for sophisticated windowing operations acknowledges the need for time-based analysis. This design facilitates the efficient processing of data within specific time intervals, catering to diverse analytical requirements.

Chapter 7

Similar or competitive Middleware

In the landscape of distributed systems, Apache Flink has several counterparts, each with its strengths and weaknesses, two of them could be the following ones:

7.1 Apache Spark

Processing Model: Initially focused on batch processing, with micro-batch based streaming capabilities.
Latency: Moderate latency due to micro-batch streaming model. Fault Tolerance: Uses data replication for fault tolerance.
APIs: Extensive support in Java, Scala, Python and R, facilitating adoption.

7.2 Apache Storm

Processing Model: Designed specifically for real-time data processing, based on data streams.
Latency: Low latency, prioritizing real-time event processing. Fault Tolerance: Guarantees fault tolerance through at-least-once processing.
APIs: Offers Java APIs and allows the creation of data processing topologies programmatically.

7.3 Key Differences

Processing Approach:
Flink: Real-time native processing.
Spark: Initially batch, with micro-batch based streaming capability.
Storm: Real-time, stream-based.

Latency
Flink: Low latency.
Spark: Moderate latency due to micro-batch model.
Storm: Low latency.

Fault Tolerance
Flink: Robust checkpointing system.
Spark: Data replication.
Storm: At-least-once guarantee.

Ecosystem
Flink: Growing, strong in event processing.
Spark: Broad ecosystem, diverse libraries.
Storm: Less diverse by comparison.

Chapter 8

References

[1] Friedman, E. , Tzoumas, K. (2016). *Introduction to Apache Flink.*

[2] Saxena, S. , Gupta, S. (2017). *Practical Real-time Data Processing and Analytics* Shannon Cutt.

[3] https://www.linkedin.com/pulse/introduction-apache-flink-shanoj-kumar-v/

[4] https://flink.apache.org/what-is-flink/use-cases/

[5] Rabl, Traub, Kastsifodimos, Markl, *Apache Flink in Current Research* De Gruyter.

[6] https://www.redhat.com/en/topics/middleware/what-is-middleware

[7] https://nexocode.com/blog/posts/what-is-apache-flink/

[8] Huesce, F. , kalavri, V. (2019) . *Stream Processing with Apache* O'REILLY.

[9] https://phoenixnap.com/kb/apache-storm