October 18, 2023

Apache Flink (v1.17.1) Conceptual Architecture Report

AUTHORS: (Team Debeggars)

Alain Ballen - alain612@my.yorku.ca
Arjun Kaura - arjunka@my.yorku.ca
Davyd Zinkiv - dzinkiv@my.yorku.ca
Nilushanth Thiruchelvam - nilut@my.yorku.ca
Peter Lewyckyj - peterlew@my.yorku.ca
Xu Nan Shen - jshenxn@my.yorku.ca

ABSTRACT

We find that Apache Flink (v1.17.1) uses pipe-and-filter as the overall architecture, with references to other architectures and mechanics, including repository style used for Flink's state backend, Chandy-Lamport algorithm for its snapshotting mechanism, as well as a watermarking system similar to one used for Google Cloud DataFlow.

Apache's major components are the Flink Application, JobManager and TaskManager, which form a core one-way data stream for input and out of data.

Apache Flink's design focuses on maintaining parallelism across all its components, ensuring vertical and horizontal scalability, high performance and fault tolerance while processing a constant stream of data, and good evolvability. These goals are achieved through the core abstractions of stateful and timely data processing.

On the more complex end of processing implementation, the responsibility falls upon the Flink application developers. A mastery of Apache Flink's technology stack is required to achieve its full potential as a processing engine. Developers will need to understand the use of a multitude of API's that are available for use, including the CoreAPI set (DataStream/DataSet). At the most complex level, the use of ProcessFunction in order to simultaneously achieve high fault-tolerant, stateful and timely data processing is necessary. There can be situations where no perfect solution exists and implementers will have to choose between data completeness and latency level in accordance to the situation.

System implementers will also need a good understanding of Flink's architecture, especially the operators, in order to best utilize Flink's potential for parallelism and scalability in a distributed cluster.

Apache Flink at its core is a data stream processing engine, but it can also process batch data as special cases.

INTRODUCTION AND OVERVIEW

Apache Flink is a powerful open-source stream processing framework that enables continuous processing of data from multiple sources with low latency and high fault tolerance. It is designed to handle large-scale data processing and analytics in real-time, making it an ideal choice for applications that require fast and accurate data processing. Flink is built on top of a distributed dataflow engine that allows it to scale horizontally and vertically, making it suitable for a wide range of use cases.

The authors **derived this report of Apache Flink, v1.17.1 based on the official documentation available on Apache Flink's website**. External information was also used to gain better insight into the framework from blog posts authored by experts on the

community forums of Apache Flink's service provider Alibaba, AWS. Referenced materials are cited in the reference section of the report. Any conflicts of information between multiple sources are reconciled by referring only to the official documentation.

Apache Flink's core goal is to execute data flow programs in a data-parallel and pipelined manner to achieve continuous/uninterrupted data processing, analytics, and data-driven handling of events. Flink's client application receives data from sources, packages them in forms of jobs which are sent to Flink JobManager, which in turn are assigned as Tasks to TaskManager for processing and streaming to data sinks. This global architecture of Flink, which is based on the Pipe-And-Filter pattern will be examined more in detail, showing how incoming dataflows start from one or more sources, get processed and transformed by user-defined operators, and end in one or more data sinks. Examples of sources and sinks will be explored in the report to display a perspective on the flexibility of Flink. Flink's core architecture allows the framework to be easily understandable from an abstract point of view, easily testable with independent components and easily modifiable with the potential of different implementations.The core components of the Flink cluster, Flink Application, JobManager, TaskManager all align with Flink's focus on distributed parallelism, and performance focused design.

Flink's core abstractions state and time will be explored in detail in addition to how they are integrated as part of the conceptual architecture. Flink's ability to manage stateful and timely data processing contributes to consistent and continuous processing of data with low latency and high fault tolerance. They also add on to the flexibility of Flink with historical data processing, pattern recognition, Machine learning versioning etc.

This report provides an overview of the conceptual architecture of Apache Flink, including its overall structure, major components and their interactions. The goals, functions, evolvability, scalability, testability etc. of the components will be discussed. The report will also cover a few performance-critical abstractions as well as two use cases. The report will then cover each major component of the Flink architecture in detail and their features. The report will include diagrams to better illustrate the conceptual architecture, and there will be a data dictionary at the end. The conclusion section proposes some future directions for Apache Flink based on the uncovered information in the report. The authors have also included lessons learned to explain the difficulties they encountered.

What is Flink

In Apache Flink, incoming dataflows start from one or more sources (message queues, distributed logs, historic logs etc.), get processed and may be transformed by user-defined operators, and end in one or more sinks (Event logs, databases, applications etc.).
Its' **core goal** is to execute dataflow programs in a data-parallel and pipelined (task parallel) manner in a way that continuous/uninterrupted data processing, analytics and data-driven handling of events can be achieved.Batch data are treated by Flink as special cases of unbounded data streams. With that in mind, implementations may differ between data streams and data sets, but the overall conceptual architecture applies to both types of data.

Control flow overview

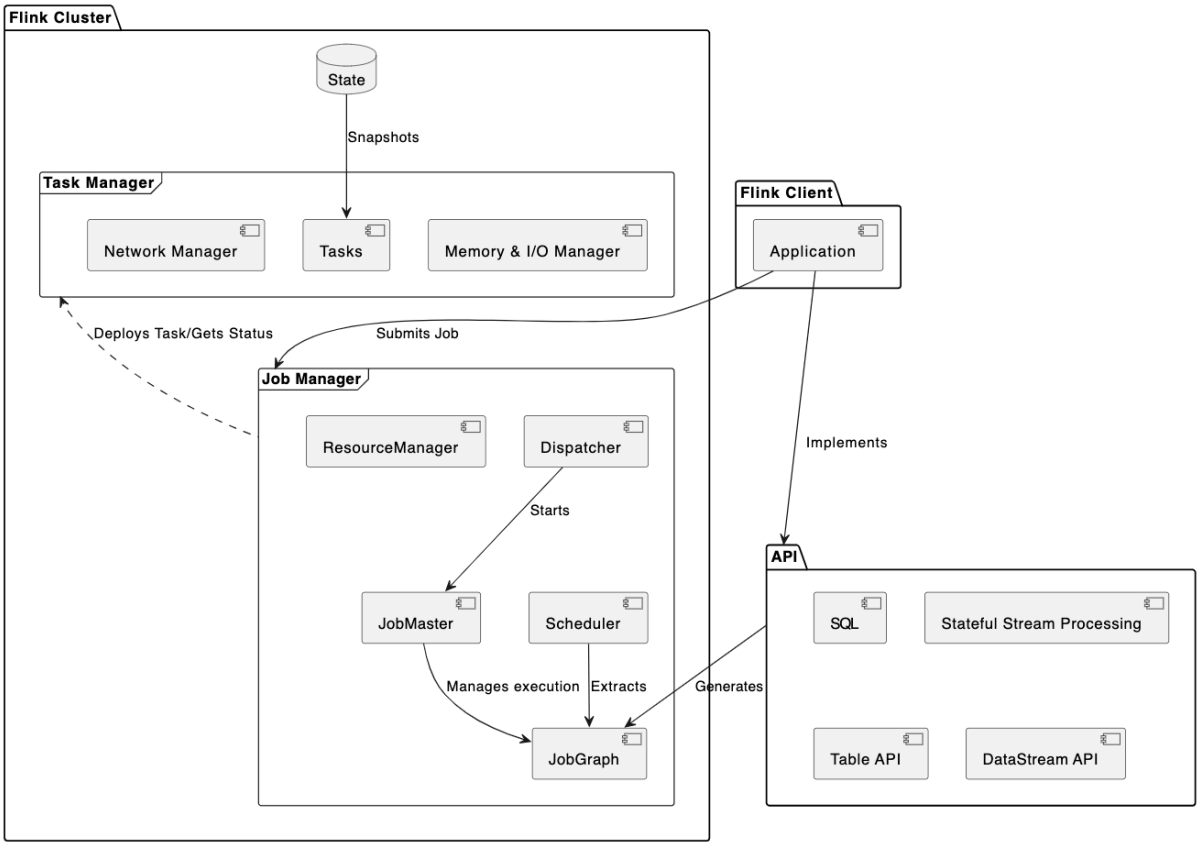


Diagram of Architecture

Flink Client Application prepares and sends dataflow to the Flink “Master” cluster. That dataflow includes the data stream to build corresponding JobGraphs. The overarching component in Flink “Master”, which is the JobManager, then breaks down the JobGraph into individual tasks and assigns them to TaskManagers. TaskManagers are connected to JobManagers announcing themselves as available whenever they are. The output is streamed to designated sinks. Task status and statistics are sent from TaskManager to JobManager and forwarded to the Client application.

Flow Interfaces

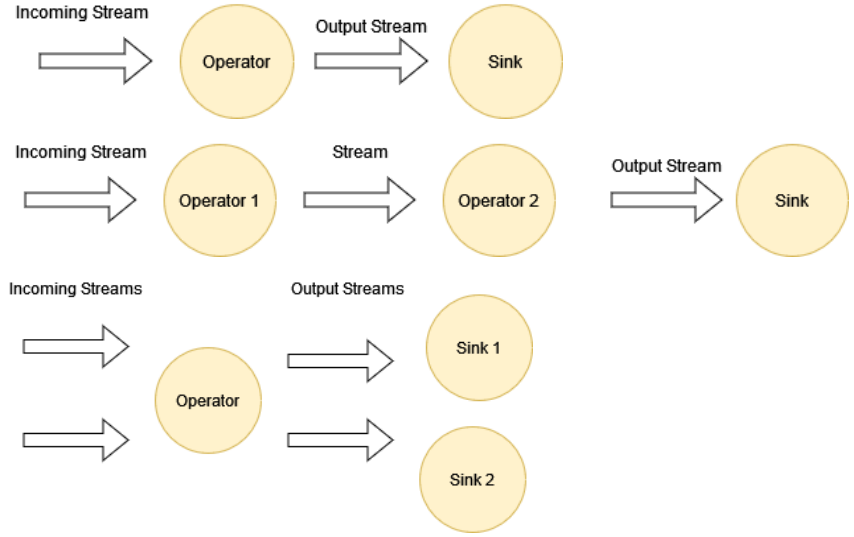
JobGraph acts as the interface between Flink Application and Flink JobManagers. They are directed logical graphs where nodes are operators and the edges define input/output relationships of the operators. Tasks (Slices of a job formed by sequences of operators) act as the interface between JobManager and TaskManager.

Flow Architecture

The overall conceptual architecture of Flink is **pipe-and-filter**. The advantages are it is easy to understand, provides easy maintenance and testing (individual components can be debugged separately), easily enhanced with Flink libraries and or external tools on either side of the pipeline without impacting the core flow. The disadvantage is that the individual filters (subcomponents) can be very complex. This architectural choice is very intuitive as Flink is ultimately a one-way data stream processing framework.

Operator

Operators are the core part of what makes up the JobGraphs. Operators are what defines the transformation that is to occur. Operators to transformation can mostly be one-to-one correspondence, there can also be transformations that consist of multiple operators. Transformations are functions programmed by developers in the Flink Application that encapsulates application logic of a program. Flink operators can also have side outputs which can be used to report errors, or other operational alerts. They also support two input streams, one for incoming data, the other for streaming in thresholds or rules or other parameters, or to simply join two streams. This architectural design allows for **complex chaining and complex operations**, since it is not up to the operators to know what happens before or after themselves. It is the responsibility of the Flink application developers to make sure the flow functions as intended.



Parallelism in control flow

Flink programs are **inherently parallel and distributed**. The Data Streams are partitioned according to their keys. The Operators are split into operator subtasks, that are independent of one another, and execute in different threads and possibly on different machines or containers. A Flink application is run in parallel on a distributed cluster. The various parallel instances of a given operator will execute independently, in separate threads, and in general will be running on different machines. Flink allows concurrency and parallelism across every individual component and subcomponent, **greatly enhancing scalability and performance**, which are necessary given the large amount of data Flink is designed to process with low latency.

Core Architectural Abstractions

State

Flink allows for stateful stream processing. Stateful operations are able to remember information across multiple events which allows Flink applications to detect patterns, aggregate events, compute versions of Machine Learning models, manage historic data etc. Flink maintains unique states and partitions streams accordingly using keys. Keyed states are distributed to stateful operators along with the corresponding keyed streams.

Keyed states are grouped into key groups; there are exactly as many key groups as defined maximum parallelism of the operators. The set of parallel instances of a stateful operator is effectively a sharded key-value store. Each parallel instance is responsible for handling events for a specific group of keys. And these states are always accessed locally to maintain high throughput and low-latency. These characteristics allow Flink states to be **vertically and horizontally scalable,** (adding local disk storage, redistribution based on cluster size).

Fault Tolerance

The Flink application is designed to be fault-tolerant through a built-in system. Known as checkpointing, the system periodically takes and stores snapshots of the application's state in another system such as a distributed file system. The key benefits of this system is that the results remain consistent and if one machine breaks, there will be no loss of data.
Snapshots are stored into two types of storage, a distributed file system and the JobManager Heap. The main difference between the two storage systems is use for experimental or production purposes. Flink uses a slightly modified version of the Chandy-Lamport algorithm when performing a state snapshot. The goal of this algorithm is to capture messages that are in transit between each node of the job graph. Including snapshots of these messages in transit would capture a consistent global state of the whole system. The modification that Flink makes is how the application captures these messages in transit. Flink uses the checkpoint coordinator which is also part of the job manager to instruct a task manager when to begin a checkpoint. Each checkpoint automatically takes snapshots of the application's state. It is incremental and optimized when it is needed to be restored quickly. Snapshots can also be triggered manually by the user and can be through an API call.
Flink follows a concept of **Exactly Once** for fault-tolerance, meaning all data is only processed no more and no less than one time. This ensures consistent results.
Flink's State-backend abstractly follows a **repository** style architecture. It helps allow the application to store large amounts of states while keeping it easily manageable. Performance is less of a concern as the accessing of data from the repository is not frequent.

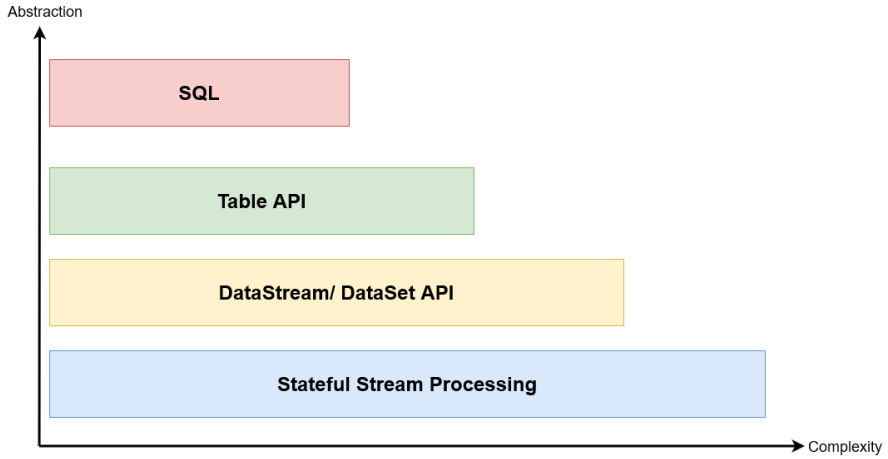
Time

Timely Stream processing is an extension of stateful stream processing. It is made possible by event timestamps recorded in the data stream, rather than using clocks of the machines processing the data. Timely Stream Processing allows for accurate re-processing of historic data using the same live data processing code. It preserves the order of events in which they occurred rather than in which they are delivered/processed; however, it cannot be guaranteed that data will arrive sorted by time. Flink allows for implementation of watermark to address that issue. Client-implemented watermark deals with event time issues of out-of-order, causing latency. (This uses the same concept as the Dataflow model used by Google Cloud dataflow). Watermarks are periodic markers that define when to stop waiting for earlier events. When an operator receives a watermark of a specified timestamp, it will update its live event time to that watermark and no longer wait for any events earlier than the specified timestamp. There can be multiple approaches to the implementation of watermarking, in most scenarios a fixed delay suffices (e.g. a 1 minute wait-window).
How developers decide to implement watermarking is essentially a decision between **latency and completeness**. The longer wait window a watermark allows for, the more complete a dataset is, but also the higher latency. Vice versa is true. More complex hybrid solutions can also be implemented (initial results supplemented by later updates). From a

parallelism perspective, watermarks are generated by source tasks independently. In the case where operators take in multiple streams, its event time uses the minimum watermark.

Implementation Abstractions (From developer's perspective)

Abstraction of Flink Implementation, developers are able to implement all the above features with the following abstraction



Graph of Abstraction vs Complexity

Stateful Stream Processing- with Process Function embedded into DataStream API. Allows free processing of events from one or multiple streams while providing consistent, fault tolerant state. Most sophisticated level of computing.
CoreAPI level- DataStream for unbounded stream, DataSet for bounded data sets. Common Data processing such as specified transformations, joins, aggregations etc.
Table API - Extended relational models. Tables with schema attached are similar to ones in relational databases. Operations such as select, join, group-by etc. Less expressive, it focuses on what to do rather than how to do.
SQL - highest abstraction. Represented as SQL queries.

Flink Application

A Flink Application is the actual Java application that one develops using the Apache Flink framework. This application can include one or multiple Flink jobs that are submitted by application for execution.

Flink Job

A Flink Job is the runtime representation of a logical graph, also referred to as a JobGraph. As described previously in the report, the logical graph is a directed graph where the nodes represent operators (e.g., map, filter, window) that perform specific tasks on the data, and edges represent data streams that connect these operators. Submitting Flink jobs is typically done by calling the execute() method on an execution environment within the Flink application. Whether the execution of these jobs happens in a LocalEnvironment (local JVM) or on a RemoteEnvironment (remote setup of clusters with multiple machines), the ExecutionEnvironment provides methods to control the job execution (e.g. setting the parallelism) and to interact with the outside world. The jobs of a Flink Application can either

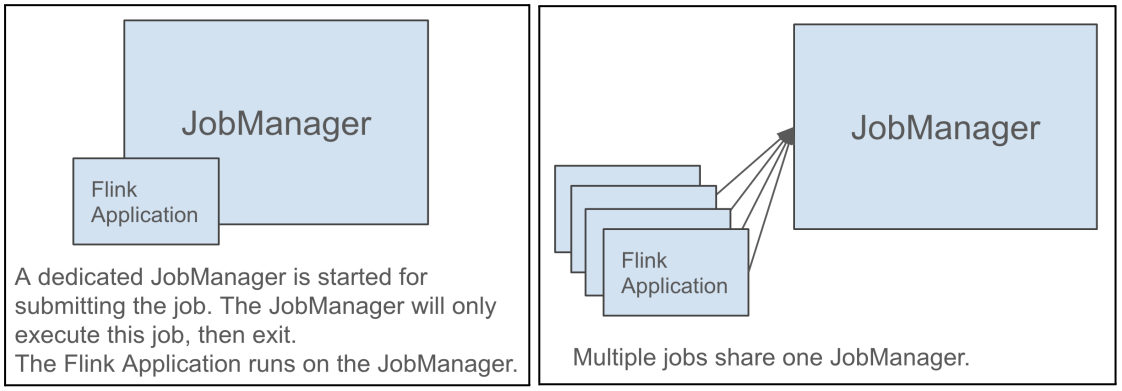
be submitted to a long-running Flink Session Cluster or a Flink Application Cluster (the third cluster, Flink Job Cluster, is deprecated and therefore is not discussed in this report). Both clusters consist of JobManager and one or more TaskManagers but differ in their lifecycles and in guarantees of resource isolation.

Flink Application Cluster

A Flink Application Cluster represents a dedicated Flink cluster that is exclusively used to execute jobs from a single Flink Application. This simplifies job submission to a one-step process. The user packages the application logic and dependencies into an executable job JAR and the cluster's entry point, known as ApplicationClusterEntryPoint, is responsible for extracting the JobGraph. This means there is no need to manually start a Flink cluster. The Application Cluster lifecycle is directly bound to the lifecycle of the Flink Application, meaning when the Flink Application's job execution is complete, the Application Cluster may be automatically terminated or deactivated. It offers a clear separation of resources, with the ResourceManager and Dispatcher scoped to a single Flink Application, promoting better separation of concerns.

Flink Session Cluster

In a Flink Session Cluster, the client connects to a pre-existing, long-running cluster capable of accepting multiple job submissions. The cluster, along with the JobManager, continues running even after all jobs are completed and only stops when manually halted. Thus, the lifetime of a Flink Session Cluster is decoupled from the lifespan of individual Flink jobs. Having a single cluster running multiple jobs implies more load for the JobManager. The drawback to this architecture is that since all jobs share the same cluster, there is some competition for cluster resources; if one TaskManager crashes, it affects all jobs running on that TaskManager. On the flip side, having a pre-existing cluster saves a considerable amount of time applying for resources and starting TaskManagers. The session cluster is valuable in situations where jobs have very short execution times, and a lengthy startup phase would negatively impact the end-to-end user experience.



Flink Application and JobManager relationship in Application (left) and Session Cluster (right)

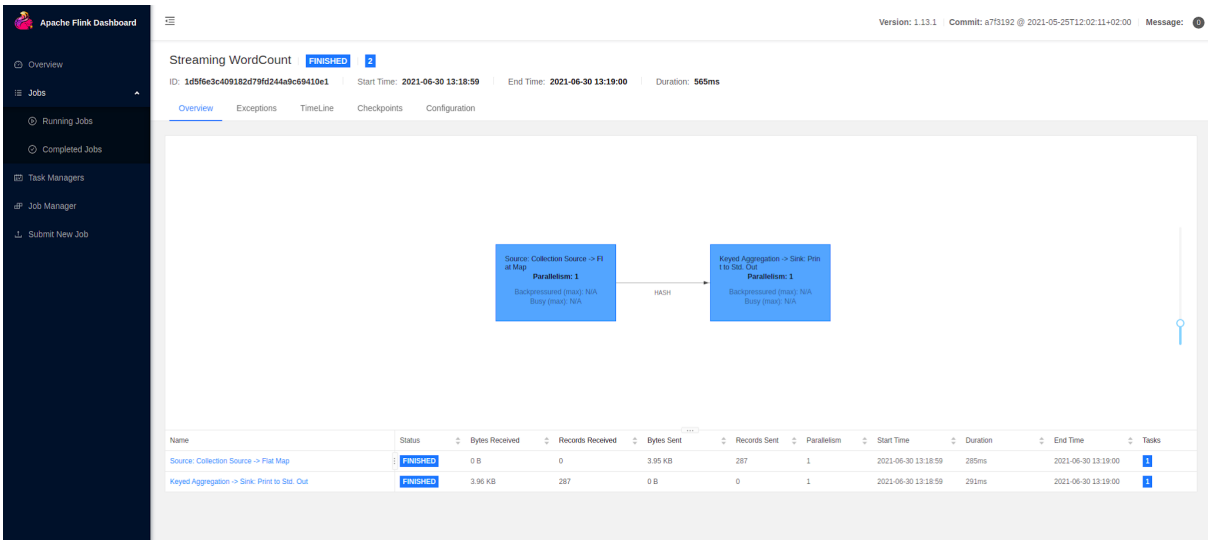
JobManager

The JobManager is the overarching process which handles the execution of an application. Each application consists of a JobGraph, a logical dataflow graph, and a JAR file that bundles all the necessary source code. The JobManager has the following responsibilities:

- The JobManager converts the JobGraph into a ExecutionGraph, which consists of tasks that can be executed in parallel.
- The JobManager makes requests to the ResourceManager for TaskManager slots and distributes the tasks of the ExecutionGraph accordingly (scheduling).
- While running jobs, the JobManager handles important tasks such as coordinating checkpoints.
- The JobManager also handles task events such as completions, cancellations or failures.

The JobManager is a single point of failure unless using a high availability deployment. By default, Flink comes with two high availability service implementations which are Zookeeper and Kubernetes. The JobManager contains three smaller sub-processes which have different responsibilities. These are the Dispatcher, ResourceManager, and TaskMaster.

The Dispatcher serves as the primary gateway for job management. It provides a REST API which is used to submit applications to be executed as well as manage ongoing jobs. The Dispatcher also provides the web interface which is accessed on `localhost:8081` by default. The interface displays information about all the running and completed jobs including job metrics, checkpoints, exceptions, and other valuable information. A dispatcher is not always necessary depending on the deployment mode.



Example of the Flink web UI
[Image source]

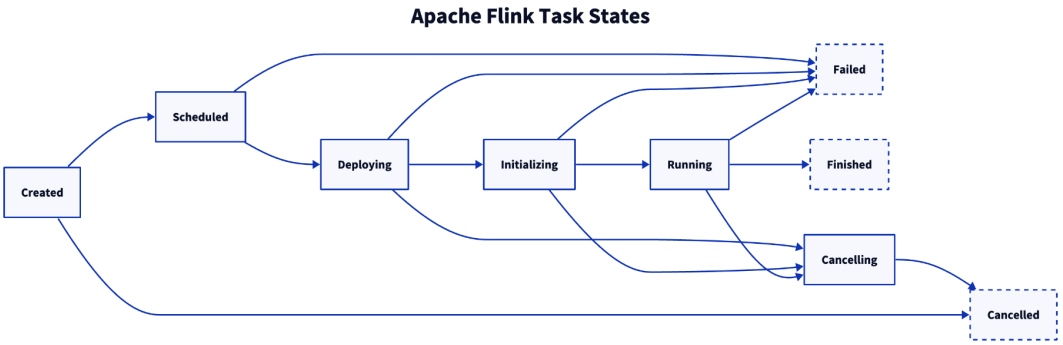
The ResourceManager is responsible for managing TaskManager slots which is Flink's unit of processing resources. When a JobManager makes a request for TaskManager slots, the ResourceManager instructs a TaskManager with open slots to allocate them to the JobManager. The ResourceManager also plays a role in resource optimization. Idle TaskManagers are terminated by the ResourceManager to free up compute resources which helps optimize the overall utilization of the cluster.

The JobMaster process simply manages the execution of a single JobGraph. Multiple jobs require multiple JobMaster processes.

TaskManager

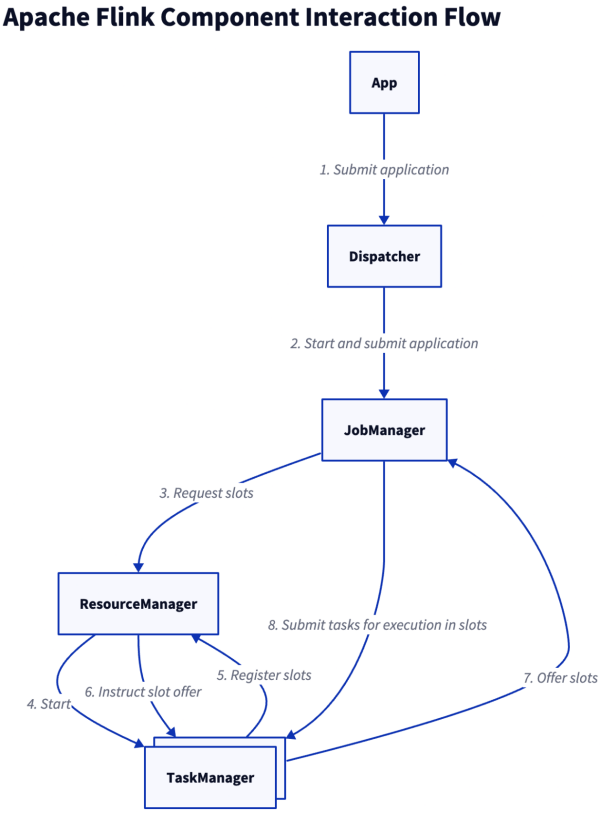
TaskManagers are the worker processes of Flink which are essential for Flink's distributed stream processing architecture. They are responsible for executing the tasks (or subtasks) of a Flink job based on the number of slots they provide. If the TaskManager has multiple slots, it has the ability to execute multiple tasks concurrently, which in turn achieves parallelism. TaskManagers are crucial for managing memory utilised by the tasks, handling data exchange between tasks, and storing state during stateful operations. In the case of any failures, the TaskManagers coordinate with the JobManager to ensure tasks are rescheduled correctly.

The following diagram showcases the different possible states a task can be in as well as their transitions. Each task begins in the “created” state and terminates in either “failed”, “finished’ or “cancelled” state.



Apache Flink Tasks State Diagram
Diagram code

At a high level, the JobManager and TaskManagers work together to execute Flink applications submitted by the client which is described in the following diagram:



Interaction of the different Flink setup components
Diagram code

The client or application submits a Flink application through the dispatcher. The dispatcher then gives the application to the JobManager which will convert the JobGraph into an ExecutionGraph. Now knowing the execution strategy of the application, the JobManager will request slots from the ResourceManager. The ResourceManger will start instances of the TaskManager if not already running, then the TaskManagers will register their slots with the ResourceManager, and the ResourceManager will instruct the TaskManagers how to utilize their available slots. At this point, the TaskManagers will offer their slots directly to the JobManager which will then submit tasks to be executed in their assigned slots. At this point, the Flink application is running.

External Interfaces:

Apache Flink has many external interfaces such as databases, socket connectors, and web services that provide APIs. It supports various external interfaces for transmitting data to and from the system as part of its software architecture. Below are some examples and explanation:

DataStream API:
Flink's DataStream APIs allow the user to stream anything that is serialized. Flink's own serializer is used for basic types such as String, Long, Integer, Boolean, Array; and composite types: Tuples, POJOs, and Scala case classes. For other types, Flink uses Kryo. It is also possible to use other serializers with Flink.

Table & SQL Connectors
Flink's Table API & SQL programs can be connected to other external systems for reading and writing both batch and streaming tables. A table source provides access to data which is stored in external systems (such as a database, key-value store, message queue, or file system).

Supported Connectors

Flink natively support various connectors. The following tables list all available connectors.

Name	Version	Source	Sink
Filesystem		Bounded and Unbounded Scan	Streaming Sink, Batch Sink
Elasticsearch	6.x & 7.x	Not supported	Streaming Sink, Batch Sink
Opensearch	1.x & 2.x	Not supported	Streaming Sink, Batch Sink
Apache Kafka	0.10+	Unbounded Scan	Streaming Sink, Batch Sink
Amazon DynamoDB		Not supported	Streaming Sink, Batch Sink

Table of supported connectors within Flink

Databases - Datastream Connectors
A few basic data sources and sinks are built into Flink and are always available. The predefined data sources include reading from files, directories, and sockets, and ingesting data from collections and iterators. The predefined data sinks support writing to files, to stdout and stderr, and to sockets.

For example:
JDBC Connectors: JDBC connectors are supported by Flink to allow for connecting to relational databases. The information includes SQL queries and data records transmitted to and from the databases.

Network Interfaces
Socket Streams: Flink can read from and write to network sockets, which could be used for custom network protocols. Information includes data transmitted over sockets.

RESTful Web Services
Flink provides a RESTful API for submitting and managing jobs. Information includes job submissions, status queries, and control commands transmitted via HTTP requests.
The monitoring API is a REST-ful API that accepts HTTP requests and responds with JSON data and is backed by a web server that runs as part of the JobManager.

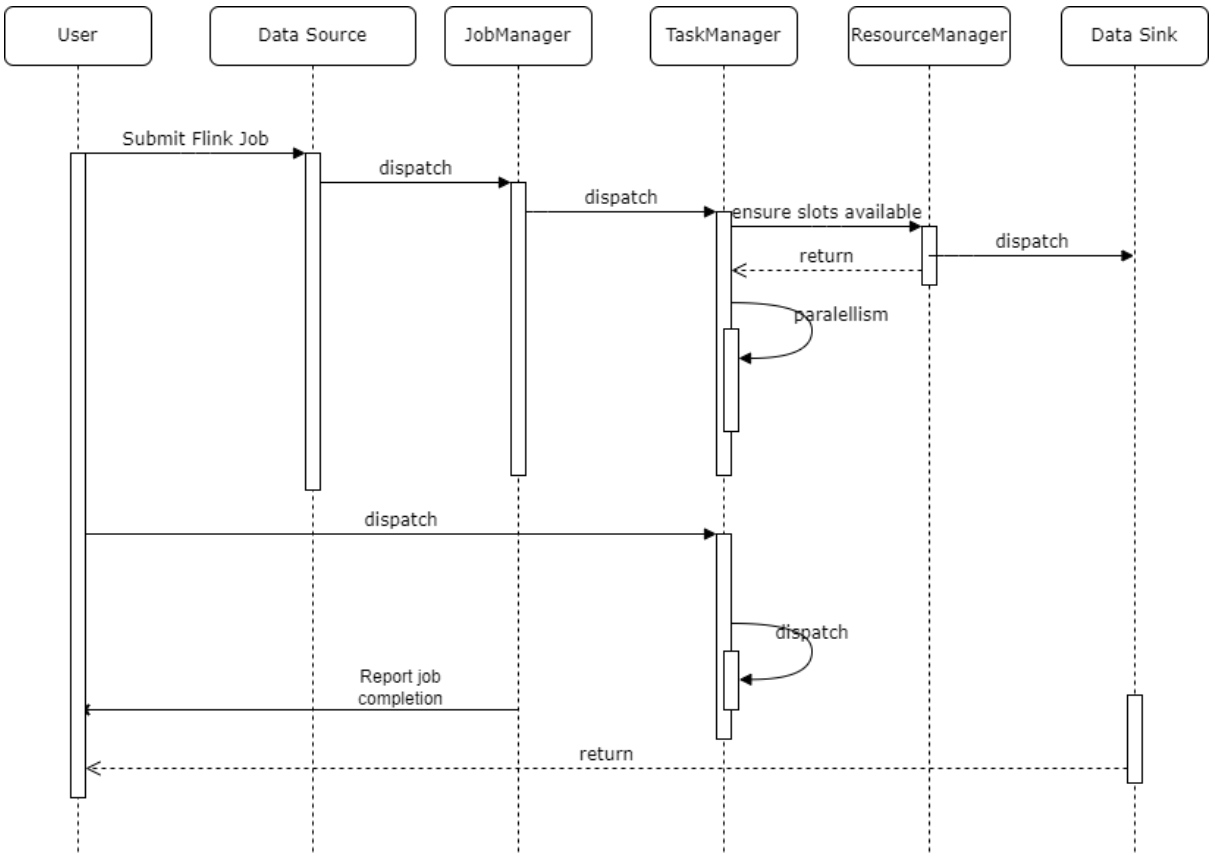
Use Cases

There are several use cases for the Flink setup. For example, consider the following conceptual hypothetical use cases that can be used using Flinks within the software development industry.

1. **Use case one:** Scaling large amounts of data for an E-commerce site.

Actors: Online users for the E-commerce site, the E-commerce site itself.
Goal: Within an E-commerce site there are an arbitrarily large number of users, i.e, 100,000. Therefore, the users generate large amounts of data through their browsing history, purchases from the E-commerce site, most visited pages, etc. This data will be used by the E-commerce site to create a friendly, effective and personalized shopping experience. In order to achieve this Flinks will be used for data processing.
Use case description and its interaction with the main Flink components.

- The online users shop within the E-commerce site and this data will be sent to the backend of the site and stored i.e, through a relational database system such as MySQL.
- This large amount of data will then be sent to Flink for data processing.
- The JobManager processes the data and delegates the workload to the TaskManager.
- The data handling is done by the TaskManager on the JVM the tasks are received from the JobManager. Within the e-commerce site, the data processing involves the categorization of purchases, analysing the browsing history/most pages visited, etc. The task manager can utilize parallelism here if there are multiple slots available for concurrency for efficient data processing speed.
- The ResourceManager ensures resource availability i.e, TaskManager slots are available and optimized. For example, within the e-commerce site, the site itself can have higher demands during peak-hours or certain holidays. Thus, even with higher demands it communicates with the JobManager to ensure allocation and deallocation of resources.
- Finally, this processed data will be sent back for personalized shopping experience for the users or even be used for content-based recommendation systems within the e-commerce site.

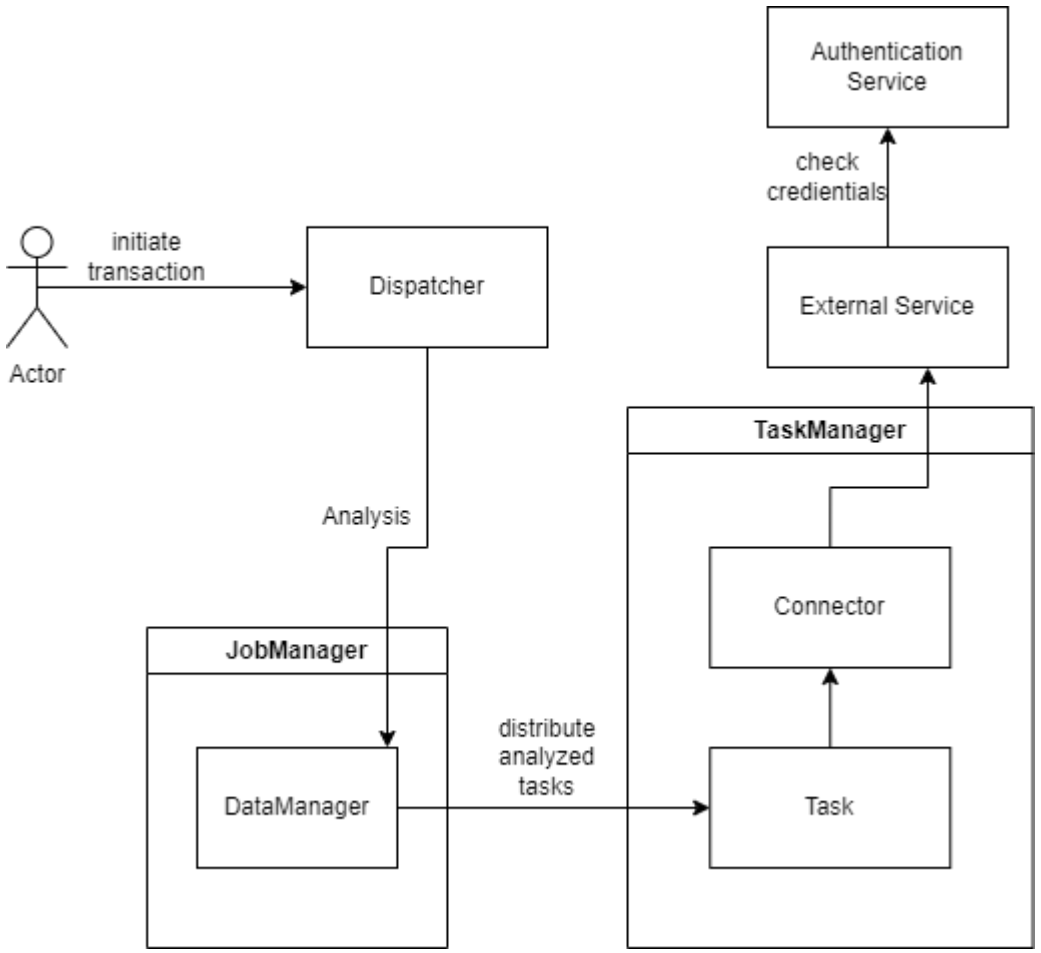


UML Sequence diagram of Use case 1

2. **Use case two:** Online payment fraud detection for online Bank/Financial systems.

Actors: Customers of the bank and the website of the Bank/Financial Institution itself.
Goal: There are arbitrarily large numbers of users within the financial institution and in order to ensure safe financial transactions a large amount of data needs to be processed to find any activities that are fraudulent, i.e, simultaneous transactions from unrecognized locations or fraudulent cheques that are deposited online.
Use case description and its interaction with the main Flink components.

- The Bank customers of the financial institution will initiate transactions which then data is collected from card payments and E-transfer (i.e, location of purchase, time of purchase and details of a merchant).
- This large amount of data will then be sent to Flink for data processing.
- The Dispatcher monitors the data then sends it to the JobManager for analysis.
- JobManager distributes the analyzed tasks to the TaskManager for each financial transaction.
- The data is going to be handled by the TaskManager. Since fraud is being analyzed the Bank customers' previous locations and cheques can be used to consider the new transactions. Existing fraudulent patterns can also be used to analyze this.
- The ResourceManager ensures there are TaskManager slots available and optimized for peak-hours of the Bank online or certain holidays.
- Finally, this processed data will be sent back to the online bank institution which can block the account user in case of fraudulent cheques or declining the new payment/e-transfer and needing more verification.



UML diagram of Use case 2

Data Dictionary

Flink Application: A Flink application is developed using the Apache Flink framework. This application can include one or multiple Flink jobs that are submitted by the application for execution.
Flink JobManager: The JobManager is the orchestrator of a Flink Cluster. It contains three distinct components: Flink Resource Manager, Flink Dispatcher and one Flink JobMasterper running Flink Job.
JobGraph: Acts as the interface between Flink Application and Flink JobManagers. They are directed logical graphs where nodes are operators and the edges define the input/output relationships of the operators.
State Backend: For stream processing programs, the State Backend determines how its state is stored on each TaskManager (Java Heap of TaskManager or (embedded) RocksDB).
Flink TaskManager: the processing components of a Flink cluster. Tasks are scheduled to TaskManagers for execution.
State: Stateful operations are able to remember information across multiple events.
Time: Timely Stream processing is an extension of stateful stream processing. It is made possible by event timestamps recorded in the data stream

Conclusions

In this Conceptual Architecture Report of the Apache Flink (v1.17.1) framework. We have outlined its core components, their features, and performance-critical abstractions. We determined that the global architecture of Apache Flink is Pipe-And-Filter, with references to repository style in its state subsystem. We found that Flink, across all its subsystems and components, is inherently parallel and is scalable both horizontally and vertically.

In summary, Apache Flink achieves high performance and fault-tolerant constant data processing through its well-designed architecture. In terms of future directions, Flink has the potential to introduce more batch-focused processing as well as continuously developing a suite of stable APIs and libraries. Flink can also put effort into streamlining Flink application development work and simplifying the implementation of event-driven processing and other features.

Team Debeggars will continue to explore Flink's architecture through Concrete analysis and discrepancy analysis.

Lessons Learned

Lesson 1:

Our team dove into the documentation to try and derive the Conceptual Architecture right from the beginning. The attempt turns out to be quite a challenge as our members lacked the foundational understanding and knowledge of the framework's function and direction. In future endeavors, our team members will seek guidance from online resources and or experts to grasp a general understanding of a system first before reading into the detailed documentation.

Lesson 2:

We believe that if we had a good understanding of Flink first, we could have formulated our own proposals of a conceptual architecture and think about the different options. That would have provided a good perspective of what the Flink team has chosen to implement in practice.

References:

Apache Flink documentation. Apache Flink Documentation | Apache Flink. (n.d.). https://nightlies.apache.org/flink/flink-docs-stable/