EECS 4314

Assignment 2

Concrete Architecture of Apache Flink (v1.17.1)

The 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

Table of Contents

Abstract

The JobManager, as described in official documentation, plays a critical role in this architecture by coordinating the distribution of tasks to TaskManagers and managing their execution. Visualization of derived dependencies (LSEdit) and source code analysis, however, reveal a more nuanced structure than what is traditionally depicted. This report examines the actual concrete architecture of Apache Flink's JobManager through a methodical comparison between the established architectural styles outlined in the documentation and the implementation as of 1.17.1.

Our study employs a combination of architectural visualization and runtime behaviour observations to uncover the existence of JobMasters, among other components, which indicates a potential evolution or discrepancy in the architecture. These findings suggest a shift towards a more decentralized or modular approach, deviating from the monolithic JobManager process initially described.

By presenting these architectural insights, the report aims to enhance the understanding of Flink's internal mechanisms and provide a solid foundation for future development and research in stream processing architectures.

This report will cover the derivation process of the JobManager and will look at other subsystems that interact with it. A high-level overview of the structure and functionality of the three main components that make up the JobManager will be looked at. Discussion of the interactions between the components, architectural styles, and design patterns used by the subsystem will be provided. The report will include diagrams to better illustrate the derivation process, components of the subsystem with interactions, and how specific classes communicate with each other. Finally, lessons learned after examining the concrete architecture will be given at the end.

1. Introduction and Overview

The JobManager subsystem of Apache Flink has numerous responsibilities that handle the execution of Flink's applications. These responsibilities include converting the JobGraph into an ExecutionGraph, task scheduling through making requests to allow the ResourceManager to communicate with the TaskManager, coordinating checkpoints when running jobs, and reacting to task events such as cancellations or failures. It is important to note that the JobManager is more of an abstraction than an actual process. There are three main components that make up the JobManager: ResourceManager, Dispatcher, and JobMaster. Each of these components has their own set of responsibilities and will be further discussed in the report.

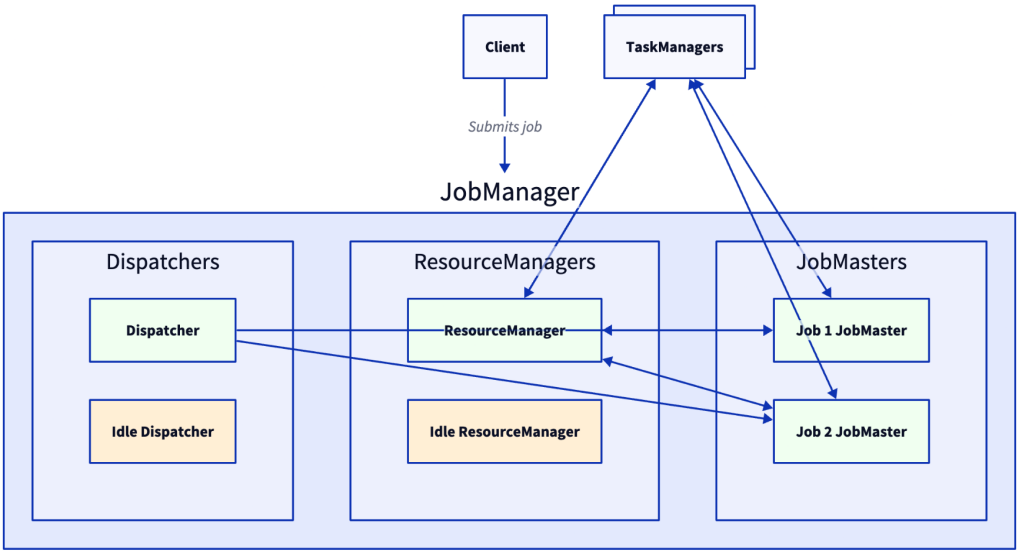

Figure 1: Flink JobManager Concrete Architecture Diagram (Diagram code)

The Dispatchers, ResourceManagers, and JobMasters cooperate to perform the tasks of an abstract "JobManager" in which the client interacts without being exposed to the more complex implementation details.

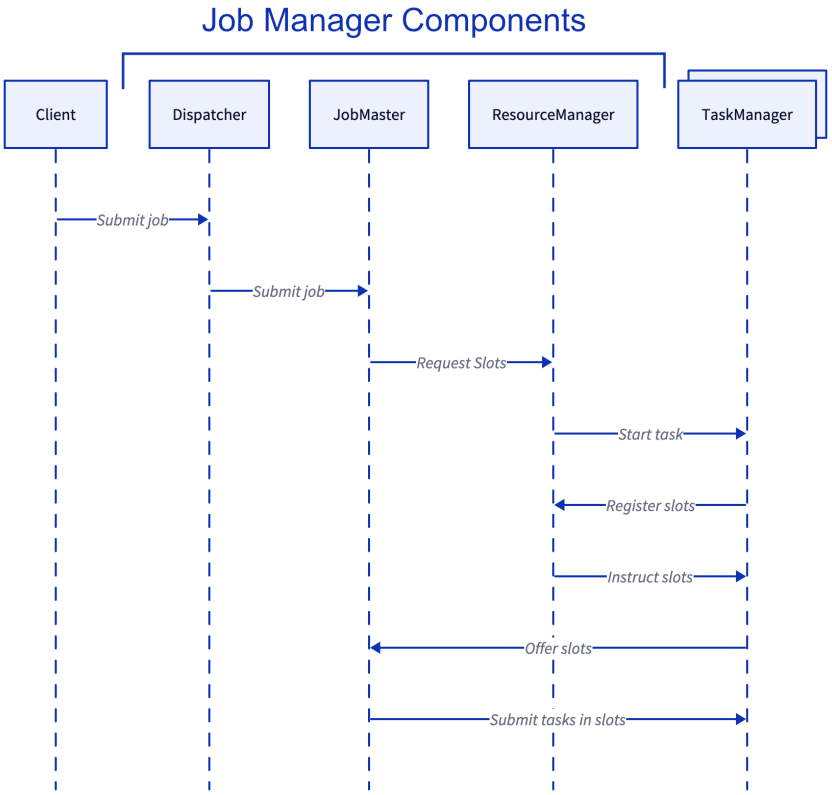

Figure 2: Flink Job Submission Flow (Diagram code)

The client remains unaware of the components and interactions involved with executing a job, and instead perceives the interaction as a 'black box' experience, with a single JobManager orchestrating the tasks.

2. Derivation Process

To extract Flink's concrete architecture we followed a formal derivation process shown in Figure 3. Using UnderstandTM software, we extracted the source code file dependencies, in a csv format. Next, we used transformUnderstand.pl script (written in Tuple-Attribute language, developed at the University of Waterloo) to convert it to raw.ta format [1]. We wrote a custom Python script to create the containment file with the subsystems we discovered, and then using JGrok, we imposed the containment on the extracted data [2]. We then use LSEdit (another system developed at the University of Waterloo) to visualize the architecture with the included subsystems [3]. We repeated the process of creating containment, imposing containment, visualizing, and analyzing to achieve the final concrete architecture; we iterated multiple times, comparing it with the abstract architecture, and looking at the source code to confirm our choices.

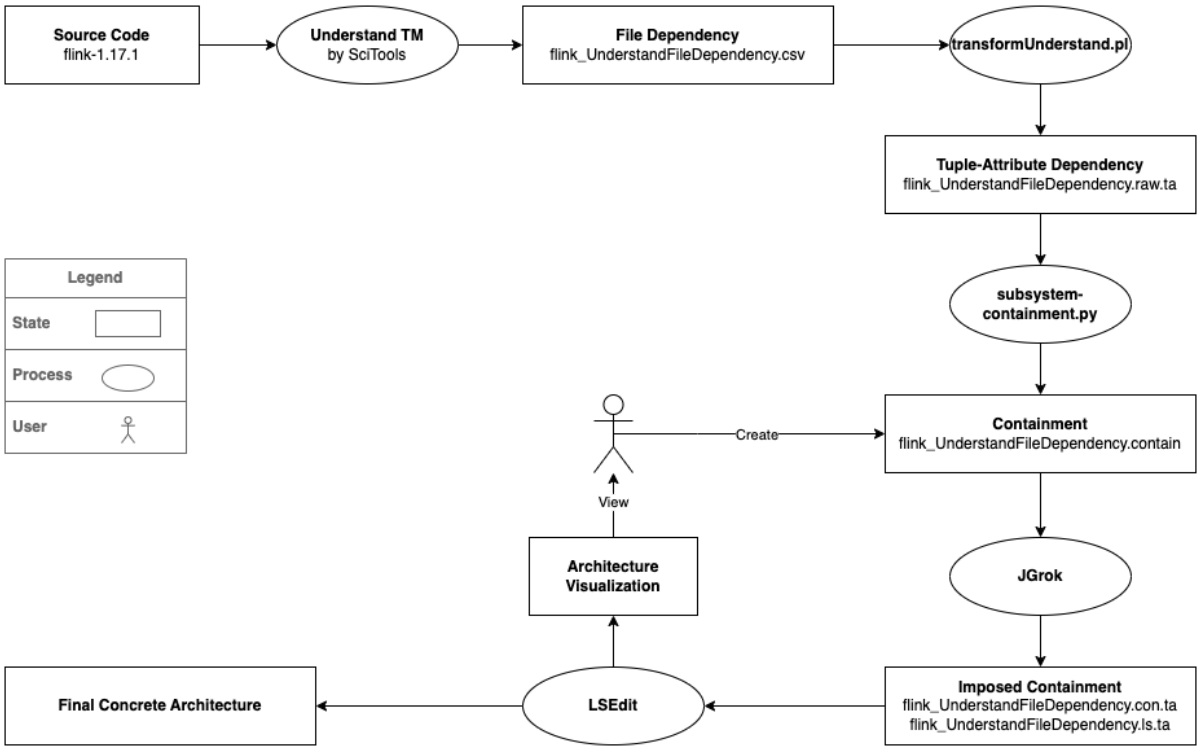

Figure 3: *Derivation process of concrete architecture.*

It is important to note that during the derivation of the architecture, we needed to reduce the amount of dependencies we worked with (step not shown in the derivation diagram). The files removed included tests, docs, examples, and walkthrough files reducing the size of the source code in half (1.5 to 0.7 MLOC).

After the first iteration (Figure 4), we had a lot of not contained file dependencies, which resulted in a diagram being illegible. It required a second iteration (Figure 5), which produced way fewer subsystems but now with very clear dependencies in between the systems such as Connectors, FlinkCore, APIs, Client, RunTime, Kubernetes, StateBeckend, and Libraries.

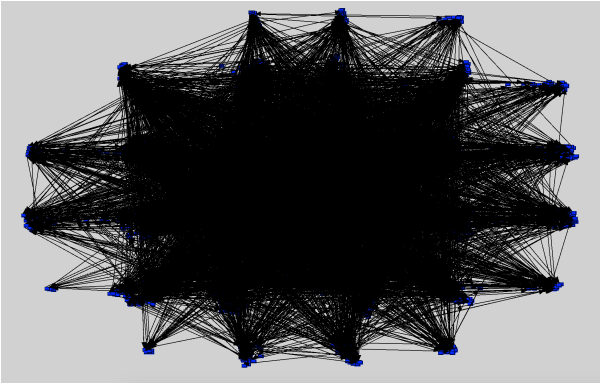

Figure 4: First iteration (full resolution)

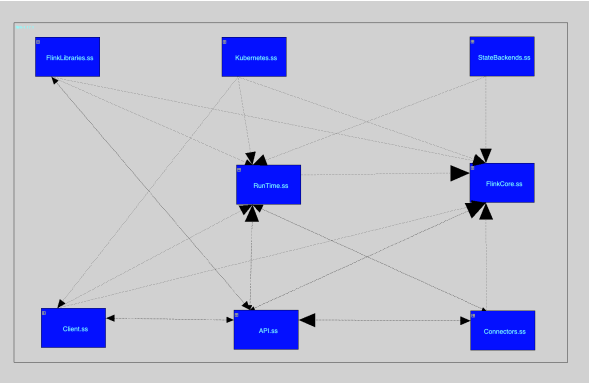

Figure 5: Second iteration (full resolution)

For a third iteration (Figure 6), we had to do some further grouping and filtering. We contained the DataStream, JavaStream, LinkTables, and StateProcessing APIs under the API subsystem, and formats and filesystems under Connectors. We removed Kubernetes

because it is not an integral system of Flink, instead, it is an orchestration system that allows for easy integration with Flink, but alternatives could also be used. The Libraries subsystem contained StateProcessing APIs and not actual libraries which we thought to be, therefore it got moved under APIs. The fourth and last iteration (Figure 7), presents a clearer picture of our subsystem of focus, the JobManager. We referred to the official Java Doc of Apache Flink v1.17.1 to identify several key abstract classes, gateways and methods and narrowed down the containment scope as well as further reducing the clinks to generate a more detailed and focused concrete architecture overview of the JobManager and its subsystems. However, it is important to note that due to the nature of the methods used here, human judgement heavily impacts the detailedness; there can be many missing dependencies that could be significant to the JobManager. To ensure accuracy, all dependencies that are shown, in addition to the division and segregation of modules and classes within different subsystems are verified through the source code and the Java Doc.

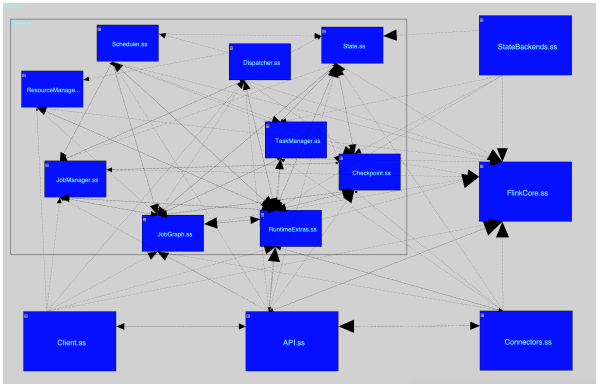

Figure 6: Third iteration (full resolution)

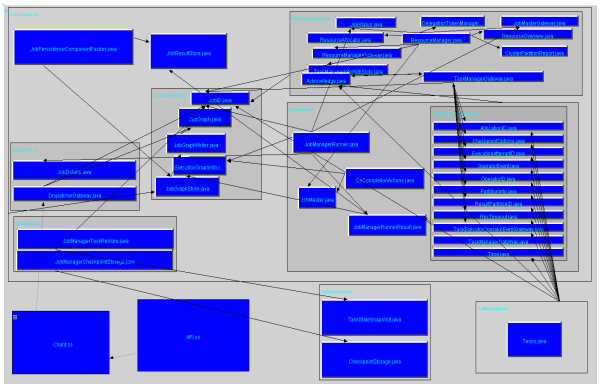

Figure 7: Fourth iteration (full resolution)

3. JobManager Overview and Subsystems

Based on the derivation results (Figure 7), we identify five core subsystems of the JobManager system: Dispatcher, Checkpoint, JobGraphWriter, ResourceManager and JobMaster. The subsystems interface with one another in order to carry out functionalities that are part of the JobManager's role in the Flink Application. The interfaces are implemented through abstract gateways as well as some generic data structures and protocols.

3.1. Dispatcher

The dispatcher subsystem is responsible for receiving JobGraphs from the API subsystem. This is done through DispatcherGateway, which is an implementation of the RestfulGateway interface that the API subsystem utilizes to send Job requests to JobManager. Once the Dispatcher receives the individual jobs from API, the subsystem then sends it to the JobGraphWriter subsystem for further processing. The DispatcherGateway also processes API calls for the checkpointing system.

3.2. JobGraphWriter

The JobGraphWriter subsystem is responsible for transforming the JobGraph class to the ExecutionGraph class. The subsystem also contains a JobGraphStore which is a service

responsible for maintaining JobGraphs in case any recovery is required. Once ExecutionGraphs have been created, they are sent from the JobGraphWriter subsystem to the JobMaster subsystem.

3.3. JobMaster

The JobMaster subsystem is a cluster of JobMaster instances. A single JobMaster is responsible for any single instance of ExecutionGraph, and JobMasters are initialised when an ExecutionGraph needs to be processed. When a JobMaster receives an ExecutionGraph, it sends a request to the ResourceManager for a task slot in order to have the ExecutionGraph processed. It is then responsible for tracking the completion or failure of the job and deciding the following actions based on the result. The JobMasterServiceProcess maintains a connection with JobMasterService which has access to the JobMasterGateway for retrieving status of the execution of a single job. Once the JobMasterServiceProcess gets the corresponding tasks statuses, through the implementation of the OnCompletionActions interface, the process is able to generate JobManagerRunnerResult reactions whether it is to notify, successfully terminate or any other actions as specified in the ExecutionGraph. The JobMaster subsystem communicates through the TaskManagerGateWay interface in order to retrieve the job statuses, and the JobMasterGateway to request appropriate task slots from the ResourceManager.

3.4. ResourceManager

The ResourceManager subsystem's main responsibility is to coordinate the distribution of individual jobs to the TaskManager cluster. It does so through a variety of information gathering implementations, including many different classes such as ResourceOverview, ClusterPartitionReport, and TaskManagerInfoWithSlots, all centring on the core ResourceManager process. The ResourceManager receives ExecutionGraph requests from the JobMaster subsystem through the JobMasterGateway interface, then based on available resources, it distributes the requests to TaskManagers with available task slots through the TaskManagerGateway interface. The TaskManagerGateway is able to process all necessary details of an individual task, including information on the specific operators which are the vectors of the JobGraphs.

3.5. Checkpoint

The checkpoint subsystem as part of the JobManager is responsible for the fault-tolerance nature of Apache Flink. It is responsible for transferring state snapshots from individual TaskManagers into a JobManagerCheckpointStorage, then forwarding the states into the configured StateBackend in the form of TaskStateSnapshot. One important distinction is that during derivation, there is another set of processes within the JobManager that is responsible for JobResult persistence. Based on class interactions and dependencies, the JobResultStore is for the most part, separate from the Checkpoint/Statebackend subsystem. While checkpoint is responsible for recording and storing task states, the Job persistence service focuses on the recovery of Job Graphs.

4. Architecture Style & Design Patterns

4.1. Pipeline and Filter

Based on the Concrete Architecture overview and the subsystem interactions, we can conclude that the overall architecture of the JobManager system is Pipe-and-filter; this conclusion is identical to the one drawn in the Conceptual Architecture Report. However, there is an observation of an abundance of dependencies between the main subsystems of JobManager. In order to prevent the dependencies from impeding the decoupling of the individual subsystems, complex inheritance structures and abstract interfaces, such as the TaskManagerGateway and the DispatcherGateway are implemented to minimize inter-reliance. There is also observation of new “pipes” created using existing “filters”, the Dispatcher subsystem which is the filter for receiving and forwarding JobGraphs added an implementation of API’s for triggering checkpoints since Flink 1.15. In this architecture, the JobManager is able to be maintained and tested separately, with each subsystem implementing their own logging and testing classes. There is also flexibility with the usage of individual “filters” for future development.

4.2. Network Architecture

The Job Graph component is an example of Network architecture. The JobMaster, which is a responsibility of JobManger, uses Job Graphs, and, as a result, multiple jobs can run simultaneously in a Flink cluster, each having its own JobMaster.
It is also referred to as a logical graph in the documentation and is a directed graph where the nodes are Operators and the edges define input/output-relationships of the operators and correspond to data streams or data sets. Similarly, networked architectures are achieved by abstracting the design elements of a network into nodes and connections.

4.3. Design Patterns

By looking at the subsystem interactions, there is observations of two design patterns.

Façade - the JobManager provides gateways between the dispatcher and client/api, and between ResourceManager/JobMaster and TaskManagers. These interfaces function as a façade. From the requester side, there is a single uniform interface, but on the other end, the interface has complex abstractions and often includes a high variety of implementations for processing different types of requests. For instance, based on different physical implementations as well as distinct job requests, the DispatcherGateway interface is implemented to initiate different forms of dispatchers including mini dispatchers or standalone dispatcher, which will inturn formulate different ExecutionGraphs and spawn JobMasters according to the details.

Master-Slave - the apparent usage of the Master-slave pattern can be observed with the ResourceManager/JobMaster to TaskManagers relation. The ResourceManager receives requests from the TaskManagers and distributes tasks based on available task slots. The TaskManagers then must report back the statuses of these jobs to the JobMaster.

5. JobManager Source Code Analysis

Examining the source code itself is one of the most effective ways to gain insights into the JobManager's function within Flink's architecture. Rather than passively studying files, we'll follow the code execution through two scenarios: Flink cluster startup and the submission of a Flink job.

5.1. Flink cluster startup

A common way to start a Flink cluster when developing Flink applications is to use the `start-cluster.sh` script. This will be the starting point for this investigation.Given no arguments, it will just run the `jobmanager.sh` script. The `jobmanager.sh` script is quite straightforward. Without any arguments, it will call the `flink-daemon.sh` script and pass `standalonesession` as an argument. Using the argument that was passed in by the previous script, the `flink-daemon.sh` script will start executing Java code with the `StandaloneSessionClusterEntrypoint` class as the entrypoint.

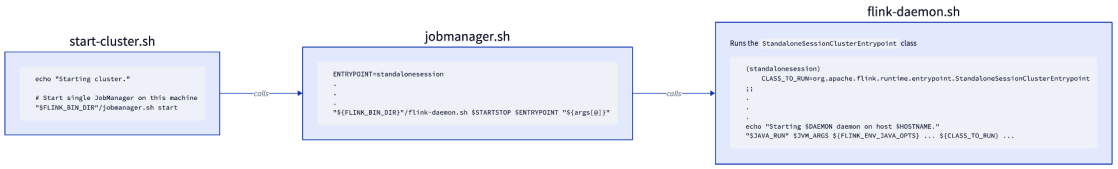

Figure 8: *Diagram detailing the execution flow of cluster startup after running the ./bin/start-cluster.sh file (Diagram code).*

`StandaloneSessionClusterEntrypoint.java` is where the Java code execution begins. It consists of a main method which contains startup checks and logging. The following code is at the end of the main method:

```
StandaloneSessionClusterEntrypoint entrypoint = new StandaloneSessionClusterEntrypoint(configuration);
ClusterEntrypoint.runClusterEntrypoint(entrypoint);
```

Following `runClusterEntrypoint()`, we can see it calls `clusterEntrypoint.startCluster()`. Through `startCluster()` we see `pluginManager` and `securityContext` get instantiated, and then a call to `runCluster(configuration, pluginManager)`. The most important code within the `runCluster()` method is the following:

```
clusterComponent = dispatcherResourceManagerComponentFactory.create(...);
```

The call to `dispatcherResourceManagerComponentFactory.create()` is noteworthy because as the name implies, it is responsible for initializing the Dispatcher and ResourceManager. Looking into the `DefaultDispatcherResourceManagerComponentFactory.java` file, we can see what happens in the `create()` method. There's a lot going on within this method (almost

200 lines), but for the sake of clarity, we'll omit everything that's not relevant to the JobManager:

```
public DispatcherResourceManagerComponent create(...) throws Exception {
 // ...
 log.debug("Starting Dispatcher REST endpoint.");
 webMonitorEndpoint.start();
 // ...
 log.debug("Starting Dispatcher.");
 dispatcherRunner = dispatcherRunnerFactory.createDispatcherRunner(...);

 log.debug("Starting ResourceManagerService.");
 resourceManagerService.start();
}
```

From this code we can see that both the Dispatcher and ResourceManager are initialised early within cluster startup. This concludes the important aspects of cluster startup with regard to the JobManager.

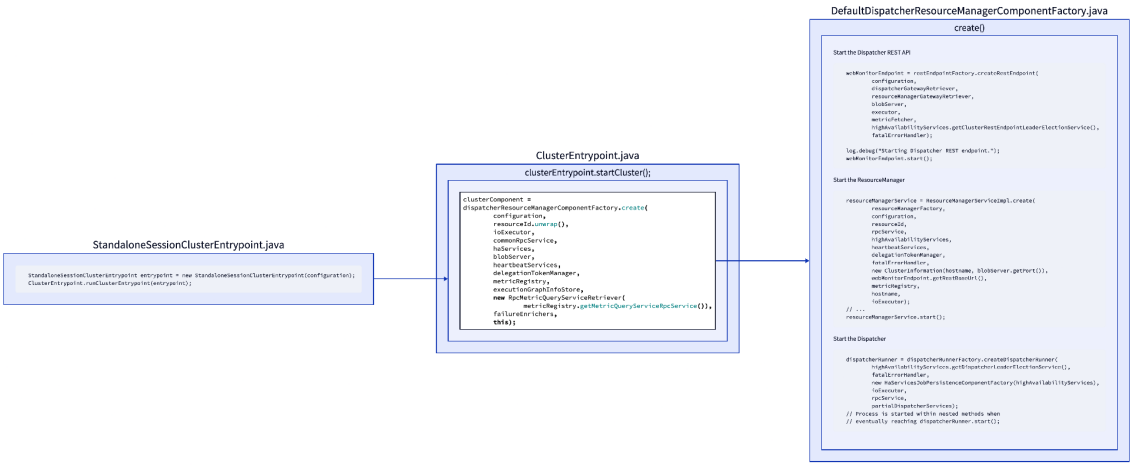

Figure 9: *Diagram detailing the execution flow of the JobManager components beginning from the StandaloneSessionClusterEntrypoint.java file (Diagram code).*

Through tracing the source code for cluster initialization, we are able to make a few observations. First, it is evident that the ResourceManager and Dispatcher are loosely coupled. They are initialised separately, and don't have a direct dependency with each other. This means that one can fail without affecting the other. Next, we can begin to see factory patterns being used within the code. This is most evident within `dispatcherResourceManagerComponentFactory`. We also encounter some asynchronous methods and patterns. For example, we see it with the `getShutDownFuture()` method which will be handled by `whenComplete()` once the state of the `clusterComponent` changes. Most importantly, we see that there is no JobManager or JobMaster being initialized. More about this will be studied in the next scenario.

5.2. Flink job submission

Tracing Flink job submission exposes a lot about the interactions between the JobManager components.

A submitted job begins in the `Dispatcher.java` file, within the `submitJob()` method which takes a `jobGraph` and a job `timeout` as arguments. This method performs a few checks on the job (whether it's in a globally terminal state, duplicate submission, etc...) and if nothing is wrong, it will pass the `jobGraph` into the `internalSubmitJob()` method. The `internalSubmitJob()` method will apply parallelism overrides to the `jobGraph` if possible, and notably call the `persistAndRunJob()` method. At the end of this method is the following line:

```
runJob(createJobMasterRunner(jobGraph), ExecutionType.SUBMISSION);
```

Before looking into the `runJob()` method, it's worth looking into the first argument - `createJobMasterRunner()`. This method is simple - it returns a call to `jobManagerRunnerFactory.createJobManagerRunner()`. As the method name implies, this creates a JobManager runner which is used within the `runJob()` method. Within `runJob()`, there's a call to `jobManagerRunner.start()`. This is where the jobMaster is finally created and started for managing the job execution.

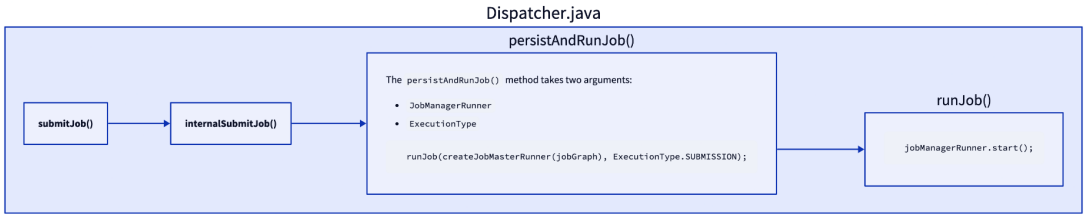

Figure 10: *Diagram detailing the execution flow of a Flink job submission through the Dispatcher.java file (Diagram code).*

We can make a few more observations based on tracing the execution of a job submission. First, we once again see factory patterns and state patterns being used. Now we also see the use of "runners" which are abstractions used to more easily manage the lifecycle and execution of components. There are "leaders" referenced throughout the code which is a strategy for improving fault tolerance where there is a leader process and ideally there will be idle processes which can replace the leader in the event of a failure. The many references to "leaders" throughout the code is just one example of the steps Flink takes to ensure redundancy and minimize the effects of failures. Finally, an important observation is that the JobMaster is not initialized right until the job is ready to run and has passed a series of checks to make sure the job is valid. There's also inconsistency between references to a "jobMaster" or "jobManager" within the source code. This is likely due to the evolution of the JobManager over years of development.

5.3. Changes to the JobManager

The JobManager has changed over time as the Flink project evolved. The most significant change to the JobManager began as part of FLIP-6 (Flink Improvement Proposal). The most relevant change is under the heading "Single Job JobManager" where it proposes the following:

> *"The most important change is that the **JobManager handles only a single job**. The JobManager will be created with a JobGraph and will be destroyed after the job execution is finished. This model more naturally maps what happens with most jobs anyways.*
>
> *Cross-job functionality is handled by other components that wrap and create JobManagers. This leads to a better separation of concerns, and a more modular composability for various cluster managers."*

Unfortunately the current Flink documentation remains misleading on the details of the JobManager. As of Flink 1.17.1, the documentation still states: "The Flink runtime consists of two types of processes: a *JobManager* and one or more *TaskManagers*." and beneath the statement is the following diagram:

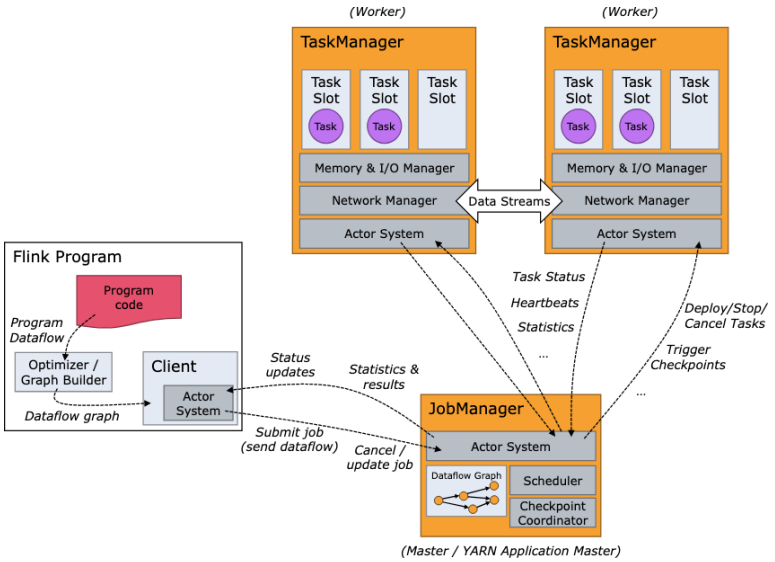

***Figure 11:** Flink's documentation showing the operation of JobManager*

Since the 1.0 release of Flink over 7 years ago the diagram and the concept of the two main processes has not been changed in the documentation. It currently follows up by mentioning that the JobManager consists of the ResourceManager, Dispatcher, and JobMaster, although by looking at Figure 11 and the previous statement, it's easy to be misled into thinking the JobManager itself is a literal process, when it no longer is. Looking at the 1.0 release of Flink, you can see the source code contains a large JobManager.scala file which was removed in a commit from early 2019. This file corresponds well with the diagram, and represents a physical process which starts with the cluster and handles job submission, slot management, and scheduling. The source of the current discrepancies is made more clear through a Flink JIRA issue (FLINK-4344) which states the following:

> *"Because of the breadth of changes, we should implement a new version of the* `JobManager` *(let's call it* `JobMaster`*) rather than updating the current* `JobManager`*.*

> *That will allow us to keep a working master branch. At the point when the new cluster management is on par with the current implementation, we will drop the old* `JobManager` *and rename the* `JobMaster` *to* `JobManager`*."*

As of Flink 1.17.1, there is indeed a JobMaster and no clear JobManager besides utility files and references to a JobManager such as the `JobManagerRunner` interface which is implemented by `JobMasterServiceLeadershipRunner`, or configurations which are still referenced as JobManager configurations which are used by the Dispatcher, ResourceManager, and JobMaster. Despite all this, a look at the source code makes it evident that the JobManager now lives as an abstraction over a more robust and fault-tolerant system composed of the Dispatcher, ResourceManager, and JobMaster which work together to handle the responsibilities of what once was a single complex JobManager process.

6. Comparison With Conceptual Architecture

The conceptual architecture of Apache Flink's JobManager, as depicted in the project's official documentation and other online resources, positions the JobManager as a key component of a Flink application. It assumes a monolithic design wherein the JobManager is responsible for the comprehensive management of job lifecycle, task distribution, resource allocation, and coordination for checkpointing and recovery. It serves as the sole orchestrator within the cluster, interfacing with multiple TaskManagers that execute the tasks.

However, after examination of Apache Flink's current architecture and source code, it reveals a more intricate and evolved structure than the monolithic one described. The introduction of the JobMaster component displays a shift towards a more decentralized model. Instead of a single JobManager handling all aspects of execution, there are now individual JobMasters dedicated to each job, supported by a Dispatcher and ResourceManager.

Furthermore, the actual implementation points to a detailed approach to coordination, which spreads the responsibility across different components. This division of labour not only enhances the system's resilience to failures but also provides a foundation for a more scalable architecture. Modern distributed systems favour modularity, allowing them to be more fault-tolerant and maintainable throughout development.

The comparison between the documented JobManager architecture and its real-world implementation in Apache Flink reveals a development that trends towards modern distributed systems principles. This evolution impacts system performance in a positive way as it makes Flink more adaptable and suitable for typical cloud-native environments. For developers, architects, and users of Flink, this understanding is crucial. It emphasizes the importance of having precise documentation that provides a true reflection of the system's capabilities.

7. Lessons learned

Lesson 1: Large software systems require a significant amount of effort to be put into pre-processing of source code/dependencies to retrieve a helpful concrete architecture that is not too detailed with an abundance of dependencies that do not contribute meaningfully to the architecture. Sometimes human judgment is necessary. As a result, verification with experts and official documentation as well as the source code are crucial to ensure the accuracy of the concrete architecture.

Lesson 2: Completing the Conceptual architecture study first proved to be immensely beneficial. This iterative process facilitates thoughtful reflection on various developmental choices. By having a clear grasp of the conceptual architecture, we can uncover unforeseen dependencies and additional abstractions and inheritances that offer valuable insights into the software's functionalities and design. For instance, the dispatcher's role in coordinating the checkpointing system was not apparent in the conceptual architecture, but a solid understanding of the pipe-and-filter model and a conceptual appreciation of the subsystems' purpose allows for a more profound comprehension through concrete iterations.

Lesson 3: In Flink’s case, the overall architectural designs remain consistent. However, it is clear that in some cases implementations differ from Architectural principles and ideas. It is very important to understand the motivation and the reasoning behind these deviations. In order to present a clear and precise overview of the architecture. It is crucial to discern important modules under the correct subsystems based on the differences from the conceptual architecture and or even the guiding principle.

Lesson 4: Understanding development history and roadmap is important, as it heavily impacts how classes, modules and even naming are implemented in the software. The architecture, as a result, is constantly shifting and changing with each version; sometimes, there can be incomplete transitions. Without a good understanding of these circumstances, a retrieved concrete architecture can be misguided, or unnecessary confusion and waste of effort can be unavoidable. In our case, the different naming of the JobMaster subsystem across different versions of Flink caused a lot of uncertainty until we grasped a good understanding behind the change and was able to have an idea of the shift.

8. References

[1] Richard C. Holt. An Introduction to TA: the Tuple-Attribute Language, technical Report, University of Waterloo, 1997. https://plg.uwaterloo.ca/~holt/papers/ta-intro.htm

[2] Grok/jGrok Documentation. https://www.swag.uwaterloo.ca/jgrok/index.html

[3] LSEdit: The Graphical Landscape Editor. http://www.swag.uwaterloo.ca/lsedit/index.html