Apache Flink

Concrete Architecture

Group Small:

Muhammad Omar
Ruth Bezabeh
Nishiket Singh
Rathul Momen
Corneille Bobda
Bartolomeo Gisone

Abstract

This report offers a comprehensive exploration into the architecture of the resource manager, a critical top-level subsystem embedded within the Apache Flink framework. The resource manager's significance lies in its role as a pivotal component responsible for orchestrating and optimizing resource allocation within the broader software system. This study is an in-depth analysis focused on unravelling the underlying architectural styles and design patterns utilized in the construction of the resource manager, aiming to provide a detailed understanding of its internal mechanisms and functionalities.

The report presents the derivation process of the system. More specifically how the different containment, dependency, Python scripts, and source codes files were used to discover how the different subsystems communicate between each other. The report also presents the concrete architecture of the Apache Flink system with its subsystems. It outlines the interactions between the different subsystems and gives an idea of how they work together.

The architectural exploration then delves into the structural framework of the Apache Flink system. It scrutinizes the architectural styles employed, highlighting their impact on system behavior and performance. Various design patterns are dissected and evaluated for their efficacy in ensuring optimal system performance and resource utilization.

Furthermore, the report compares the Conceptual Architecture to the Concrete Architecture and highlights the absences, divergences, and convergences between the two. It also gives a detailed description of the Resource Manager subsystem and the different sub-subsystems it contains.

In conclusion, this report not only presents a detailed investigation into the architecture of the resource manager in Apache Flink but also serves as a critical contribution to the overall comprehension of the system's architecture. Its insights serve as a valuable resource for software engineers and system architects seeking to understand and optimize resource management strategies in distributed computing environments.

Report Organization	
Introduction	This section will provide the purpose for the report, the report organization, and salient conclusions.
Concrete Architecture	This section will provide a visual representation of the top level concrete architecture of Apache Flink, as well as a discussion on the concrete architecture.
Derivation Process	This section will provide a detailed explanation of the derivation process the team used to come to generate the concrete architecture. This section will also discuss any alternative processes that were deemed unfit.
Subsystems and Interactions	This section will reiterate subsystem functionality and interaction. It will also introduce and explain any new subsystems discovered.
Architectural Styles	This section will present and explain any architectural styles discovered in the concrete architecture.
Design Patterns	This section will present and explain any architectural styles discovered in the concrete architecture.
Architecture Comparison	This section will compare the concrete architecture with the conceptual architecture from the previous report. It will discuss any inconsistencies and similarities between the two.
Resource Manager Subsystem	This section will introduce and describe the resource manager subsystem, its functionality, and its interactions with the rest of the Flink system. It will also provide the top level architecture of this subsystem.
Use Cases	This section will provide a use case and state diagram to provide the reader with a better understanding of how the resource manager interacts with the rest of the Flink system in a real world context.
Data Dictionary	The section is a glossary that defines all key terms used in this report.
Naming Conventions	This section will list any naming conventions or abbreviations used in this report. It will define the convention or abbreviation, and indicate where in the report it is used.
Conclusions	This section will summarize all key findings and any proposals for the future.
Lessons Learned	This section will highlight and explain any elements that the team wishes they knew beforehand or did differently.
References	References and Sources.

Table 1.0 Report organization

Introduction

The purpose of this report is to provide a clear and detailed overview of the concrete architecture of Apache Flink and the ResourceManager subsystem. This report will present a visual representation of the top level concrete architecture of Apache Flink, as well as the top level concrete architecture of the ResourceManager subsystem. Apart from the focus of these two elements, this report will break down and explain the subsystems and their interactions, architectural styles and design patterns, and compare the derived concrete architecture with the conceptual architecture that was generated in the previous report

Concrete Architecture

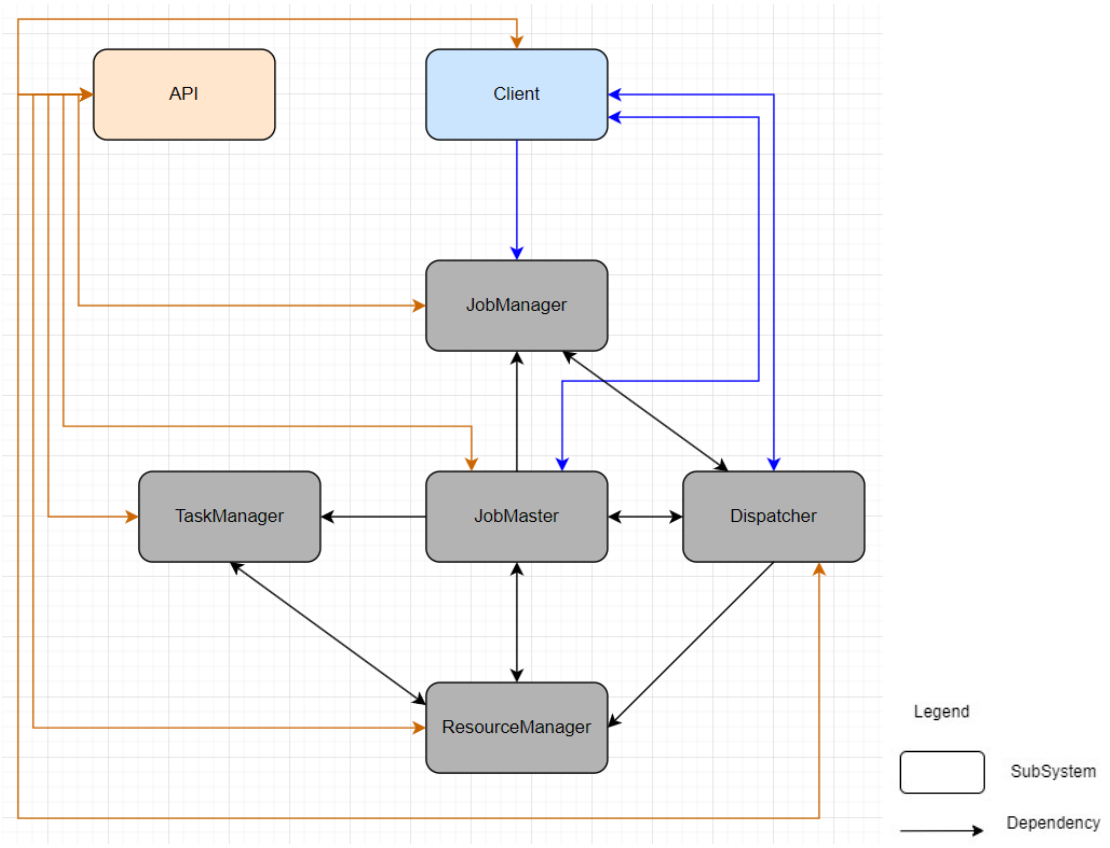

Figure 1.1 - Concrete Architecture

Above is the cleaned up extracted concrete architecture of Apache Flink. In the process of cleaning up, no dependencies were added or removed. The clean up consisted of reorganizing the subsystem links for a cleaner look. The extracted concrete architecture introduced several changes in the understanding of how Apache Flink works.

We can see how the JobManager, Dispatcher, JobMaster, TaskManager, and ResourceManager are interconnected. This disproves the assumption that there is a hierarchical communication between these

subsystems. Initially, the assumption was that the client only talks to the JobManager, the JobManager talks to the ResourceManager, Dispatcher, and JobMaster, and the services that these three provide can only be communicated to the task manager through the JobManager. This has proven to be false as we can see direct connections between the TaskManager and Resourcemanager, JobMaster and Taskmanager, etc.

We can also see how the API subsystem plays a significant role in the Apache Flink subsystem. A system like Apache Flink uses several external resources and services to perform effectively. For this reason, the API subsystem essentially communicates with all subsystems in Apache Flink. Again, this disproves the hierarchical communication assumption, where the API subsystem only communicates with the Client subsystem.

Later, this report will discuss the interactions of these subsystems, and will compare this extracted concrete architecture with the proposed architecture in the previous report to provide a better understanding of the concrete architecture.

Derivation Process

The derivation process begins with the provided files. The Apache Flink source code was retrieved from the team website, and the UnderstandDependencyFile.raw.ta files was provided by the professor and retrieved from eClass. A script, **newcontain.py** (figure 1.2) was used to generate the UnderstandDependencyFile.contain file from declared $INSTANCE lines, where it looked at directory naming/structure as well as cLinks present in the raw.ta file to derive subsystems for Flink's top level architecture, and the resource manager architecture.

```
def create_containment_file(dependency_text):
    dependency_lines = dependency_text.strip().split('\n')
    containment_lines = []
    psubsystem = ''
    csubsystem = ''
    for line in dependency_lines:
        parts = line.split()
        if parts[0] == '$INSTANCE' and parts[2] == 'cFile':
            cfile = parts[1].split('/')
            if(cfile[3] != 'test'): #ignore test directories
                for i in range(0,len(cfile)-2):
                    if(cfile[i] == 'apache'): #take whatever is after ../apache/ as a subsystem
                        if(cfile[i+1] == 'runtime'):
                            i = len(cfile)
                        else:
                            csubsystem = csubsystem + cfile[i+1]
                            file = ''
                            for j in range(0,len(cfile)):
                                file = file + cfile[j]
                                if(j != len(cfile)-1):
                                    file = file + '/'
                            #file = cfile[len(cfile)-1]
                            containment_lines.append(f'contain {csubsystem} {file}')
                            csubsystem = ''
                            i = len(cfile)
            else:
                csubsystem = 'Tests'
                file = ''
                for j in range(0,len(cfile)):
                    file = file + cfile[j]
                    if(j != len(cfile)-1):
                        file = file + '/'
                #file = cfile[len(cfile)-1]
                containment_lines.append(f'contain {csubsystem} {file}')
                csubsystem = ''


    containment_text = '\n'.join(containment_lines)
    return containment_text
```

Figure 1.2 - *newcontain.py* script

```
def parse_dependency_file(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        dependency_text = file.read()
    return dependency_text

def create_containment_file(dependency_text):
    dependency_lines = dependency_text.strip().split('\n')
    containment_lines = []
    previous = ""

    for line in dependency_lines:
        parts = line.split()
        if (parts[1] != previous):
            subsystem = parts[1]
            containment_lines.append(f'$INSTANCE {subsystem} cSubSystem')
            previous = parts[1]

    containment_text = '\n'.join(containment_lines)
    return containment_text

def main():
    # Parse the dependency data from a .raw.ta file
    dependency_file_path = "flink_UnderstandFileDependency.containment"  # Replace with your file path
    dependency_text = parse_dependency_file(dependency_file_path)

    # Generate the containment file
    containment_text = create_containment_file(dependency_text)

    # Write the containment text to a .containment file
    with open("flink.txt", "w", encoding="utf-8") as containment_file:
        containment_file.write(containment_text)

    print("Containment file has been created and saved ")

if __name__ == "__main__":
    main()
```

Figure 1.3 - *getsub..py* script

Another script **getsub.py** (figure 1.3) was created and used to determine the subsystems present in the newly created UnderstandDependencyFile.contain file which would contain all the $INSTANCE entries in their

appropriate subsystems. **newcontain.py** was then run iteratively to produce contain files for the JobManager, JobMaster, TaskManager, Dispatcher, ResourceManager. It was also modified to decompose ResourceManager into its subsystems.

The lsedit tool was used to generate a visual representation of the Flink and ResourceManager top level concrete architecture. This graphical representation was able to display all subsystems and subsystem connections. Lsedit also enabled the team to look into each subsystem and see which files are interacting with each other, and which files in the subsystem are interacting with other subsystems.

Previous Processes
The first attempt to derive the concrete architecture involved removing what the team believed to be irrelevant or unwanted files/file dependencies. This method was later abandoned as there was not enough information in the materials to make the assumption that a file or dependency was irrelevant to the concrete architecture of Flink.

The second derivation method attempted was to assign sections of the source code folder to each team member. Each member would then traverse their assigned section of the source code folder, using directories and naming conventions to determine subsystems. This method was quickly deemed inefficient and subsequently abandoned.

Subsystems and Interactions

JobManager: Responsible for coordinating and managing Flink jobs. JobManager accepts and manages job submissions, receiving jobs from the client, generating a job execution plan, and monitoring/managing the job's lifecycle

TaskManager: Responsible for executing the tasks within a Flink job, and is capable of running multiple subtasks of a job in parallel.

Resource Manager: The ResourceManager plays a key role in managing and allocating resources for Flink. It supervises the distribution of resources ensuring efficient execution of Flink jobs.

Dispatcher: The Dispatcher assists in the submission of Flink jobs. Upon a submission, the Dispatcher creates a new JobMaster for that job. It also provides information about job executions.

JobMaster: The JobMaster is responsible for managing the execution of a JobGraph. Each Flink job has its own JobMaster. JobMaster is responsible for generating an execution sequence for a Flink job. It decides the order of the Jobs operations and data exchange. It tracks the status, progress, and dependencies of the tasks in a Flink job, as well as dynamically add or remove TaskManagers.

Interactions

As seen in the concrete architecture diagram, the top level subsystems of Flink are interconnected with each other. Here are some key interactions between these subsystems.

Client to Dispatcher	The client sends the job submission requests to the dispatcher, where it then coordinates

	the job execution.
Dispatcher to JobManager	The Dispatcher sends the job to the appropriate JobManager. Once the JobManager receives the job, the Dispatcher signals to the client so it can begin monitoring the progress.
JobManager to ResourceMa	JobManager communicates with the ResourceManager to allocate the appropriate resources needed for the job
TaskManager to ResourceM	TaskManagers send periodic heartbeats to the resource manager to signal that they are functioning correctly. This helps the resource manager detect failures and reallocate resources.
ResourceManager to TaskM	ResourceManager and TaskManager communicate with each other to request and allocate task slots.

Table 1.1 Top-level subsystems and interactions

Architectural Style

Client-Server Style Architecture:

Apache Flink can be used to implement a client-server RPC pattern for distributed data processing and service communication[1]. The client initiates requests, and the server processes them, utilising Flink's features for distributed data processing, fault tolerance, and integration with external systems.

Client Request: The client component initiates the RPC by sending a request to the server. This request can be a specific task or operation that the server is expected to perform.
Server Implementation: The server component, which is a Flink job or service, is responsible for handling incoming requests. It has a defined API or method that clients can invoke remotely.
Remote Execution When the server receives an RPC request, it processes the request, executes the necessary logic, and may return a response to the client. This execution can include data processing, transformations, aggregations, or any other custom business logic.
Communication Mechanism and Data Exchange: Flink provides various mechanisms to facilitate RPC-style communication between components. This can include using Flink's built-in messaging and data transport capabilities, as well as external systems like Apache Kafka or other message brokers to exchange data between Flink applications.In many cases, the data to be processed by the server component is exchanged via Flink's data streams. Clients may send data to a Flink job, and the server processes it as it arrives. This aligns with Flink's streaming architecture.
Asynchronous Processing: RPC in Flink can be asynchronous, meaning that the client doesn't need to wait for the server to complete the operation. It can continue processing other tasks and periodically check for the result or response from the server.
Fault Tolerance: Flink provides fault tolerance mechanisms to ensure that the RPC requests are not lost, and the processing state can be restored in case of failures.

[1] "flink/flink-rpc/flink-rpc-core/src/main/java/org/apache ... - GitHub." https://github.com/apache/flink/blob/release-1.17/flink-rpc/flink-rpc-core/src/main/java/org/apache/flink/runtime/rpc/RpcEndpoint.java. Accessed 3 Nov. 2023.

Scalability: Flink's distributed and parallel processing capabilities make it suitable for implementing scalable client-server RPC patterns. You can deploy multiple instances of Flink jobs or services to handle a high volume of requests.

Integration with External Systems: Flink can easily integrate with external systems for sending and receiving RPC requests and responses. For example, you can use Flink connectors to communicate with message queues, databases, or other services.

Advantages: First and foremost, its distribution of data is straightforward, allowing for efficient and parallel processing of large volumes of data. The transparency of data location simplifies the management of distributed systems, making it easier for developers to work with and enabling seamless data flow across the cluster. Flink's ability to mix and match heterogeneous platforms ensures flexibility and compatibility with various data sources and sinks, providing a versatile framework for diverse use cases. Furthermore, it facilitates the easy addition of new servers or the upgrade of existing ones, making it highly scalable and adaptable to changing business needs. These features collectively make Apache Flink a powerful choice for real-time and batch data processing, offering the reliability and flexibility required in today's data-driven applications.

Disadvantages: Lack of a central register for names and services, which can complicate the process of discovering available services within a Flink-based system. In the absence of a centralised registry, it becomes challenging to maintain a comprehensive directory of the various services and their current status, making it difficult for developers and operators to ascertain what services are accessible in the cluster. This decentralised approach can lead to a lack of transparency and hinder efficient coordination and communication between different components, potentially causing delays in troubleshooting, monitoring, and ensuring the seamless interaction of services. In scenarios where visibility and streamlined service discovery are crucial, the absence of a central registry can pose a significant hurdle and may necessitate additional effort to address these challenges effectively.

Design Patterns

1. Factory Design Pattern:

Sources and sinks play a crucial role in ingesting and outputting data from various systems. The Abstract Factory pattern can be applied to create families of source and sink factories. These factories can produce source and sink instances tailored to different data connectors, such as Kafka, Hadoop, or database connectors. This flexibility allows Flink to adapt to diverse data sources and sinks without tightly coupling to their concrete implementations.

Abstract Factory pattern abstracts the underlying data connectors. This abstraction allows Flink to define a common interface for sources and sinks, making it easier for users to switch or extend connectors as needed. For example, it enables developers to write custom source and sink factories that adhere to the established interface, enhancing the system's extensibility.

It enables a plugin or extension mechanism within Flink, where developers can create custom source and sink factories. This feature allows users to seamlessly integrate Flink with new data systems or proprietary connectors, promoting modularity and reusability. It is applied in the design of source and sink factories and

their related connectors. This application offers advantages such as flexibility, data connector abstraction, plugin support, and integration with external systems, contributing to Flink's adaptability and extensibility in the context of data processing and stream analytics.

2. Facade Design pattern:

The Facade pattern is a structural design pattern that provides a simplified interface to a set of interfaces in a subsystem.

Apache Flink is a comprehensive framework with many components and APIs. The Facade pattern can be used to create a simplified, high-level API or "facade" that abstracts away the complexity of interacting with various Flink components. This makes it easier for developers to work with Flink, especially for common use cases, by providing a straightforward interface for common tasks. Flink users can benefit from a more intuitive and less complex interface for interacting with Flink's core components like DataStreams, Windows, and State Management. This simplification can enhance productivity and reduce the learning curve.

Flink can encapsulate repetitive and boilerplate code within the facade, eliminating the need for users to write extensive code for common tasks. This simplifies application development, reduces the chance of errors, and improves code maintainability. The simplified, user-centric facade API can lead to increased developer productivity, as it allows them to focus on application logic and data processing requirements rather than grappling with the intricacies of the framework.

The Facade pattern can encapsulate best practices and patterns for using Flink effectively. It can guide users in following recommended approaches and configurations.

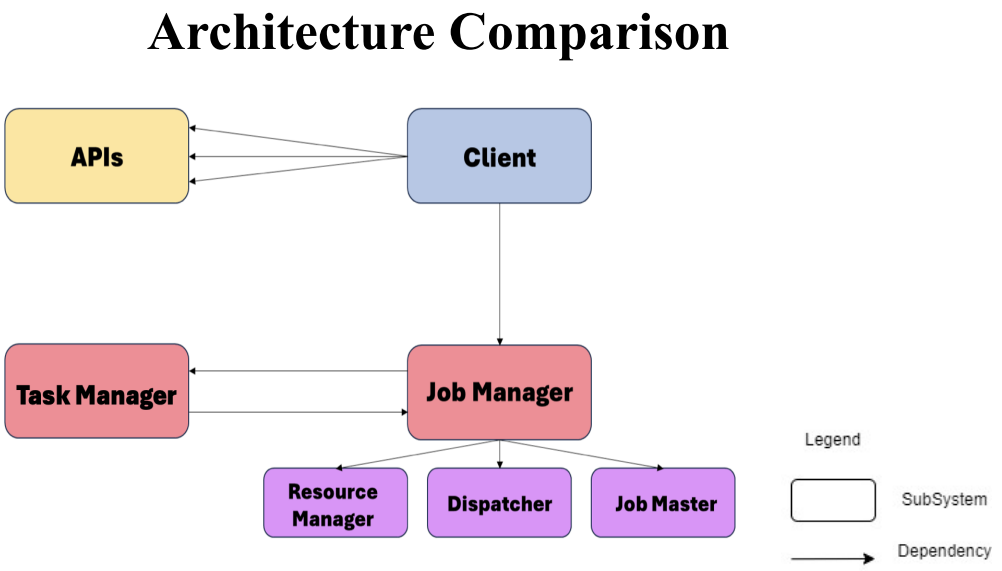

Figure 1.4 - Conceptual Architecture

Figure 1.3 shows the proposed Conceptual Architecture. The architecture had some inconsistencies compared to the Concrete Architecture presented in figure 1.4 below. The diagram was created with very limited knowledge of the system and with little information available. It did not help that the information was outdated.

Absences

- There are a few dependency relationships in the Concrete Architecture that were absent in our Conceptual Architecture. The dependency relationships between the API and every the other subsystems were not represented in the Conceptual Architecture. It was assumed that the a relationship only exists between the API and the Client Application. In the Concrete Architecture we can see that the API extends its communication everywhere in the system.
- The bidirectional dependency relationship between the Task Manager and the Resource Manager was also absent in our Conceptual Architecture. This is because we assumed the Resource Manager was a sub-subsystem within the JobManager. As seen in the Concrete Architecture the Resource Manager is a subsystem on its own and the two subsystems depend on each other.
- The role of the Job Master as well as its dependency relationships were not represented in the Conceptual Architecture. It was assumed that the Job Master had a master-slave relationship with the JobManager. A dependency relationship where the JobManagerhad more control in the entire system. As shown in the Concrete Architecture, the Job Master is the subsystem that has a more pivotal role in the entire system. The Job Master has a dependency relationship (unidirectional or bidirectional) with every other subsystem as well as with the API.
- The Conceptual Architecture does not represent the dependency relationship between the Job Master and the Dispatcher. It was assumed that the Job Master is a sub-subsystem of the JobManagerthat only interacts with the JobManager. As shown in the Concrete Architecture the Dispatcher has a bidirectional relationship with the Job Master.
- In the Conceptual Architecture the Client Application has limited interaction with the other subsystems, notably the API and the JobManager. After implementing the Concrete Architecture we realized there also exists dependencies between the Client Application and the Dispatcher and Job Master subsystems.

Convergences

- There are dependency relationships we found in the Concrete Architecture that were also represented in the Conceptual Architecture. Such as: Client-JobManager, Client-API, JobManager-Dispatcher (only one direction is represented).

Divergences

- Our Conceptual Architecture included a bidirectional dependency relationship between the JobManagerand the Task Manager, but as seen in the Concrete Architecture the JobManagerdoes not directly interact with the Task Manager. Instead, the Task Manager directly interacts with the Job Master.
- A dependency relationship where the JobManagerdepends on the Job Master was implemented in the Conceptual Architecture. There exists a relationship between the two but it is the Job Master that depends on the JobManager.

Resource Manager Subsystem

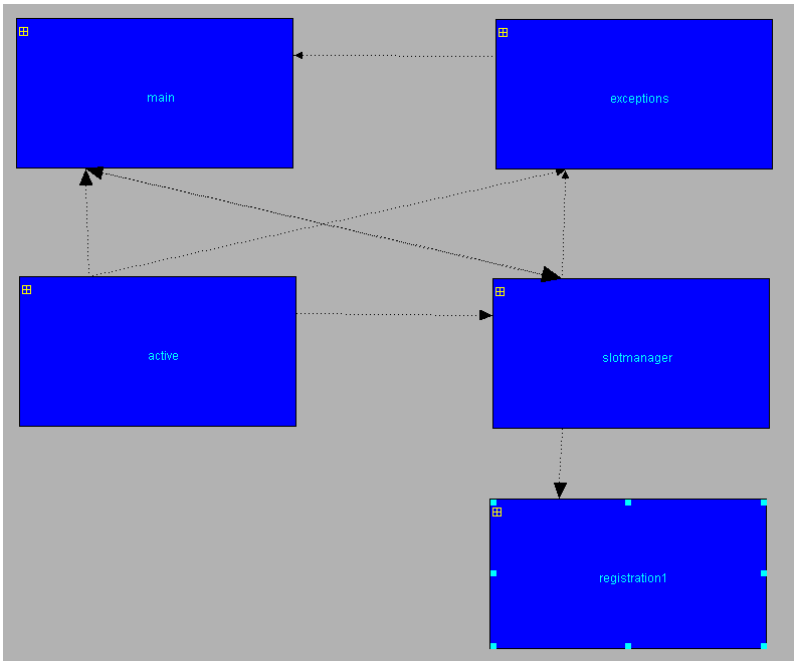

Figure 1.5 - ResourceManager top level architecture visualized in lsedit

The resource manager subsystem works in conjunction with other subsystems to support Task Managers and JobManagers to have the resources necessary to complete tasks and or to complete job graphs. The resources that the resource manager works with are task slots, which both task managers and job managers utilise. Below is a diagram from lsedit visualising how the subsystems of the resource manager interact with other subsystems in Apache Flink.

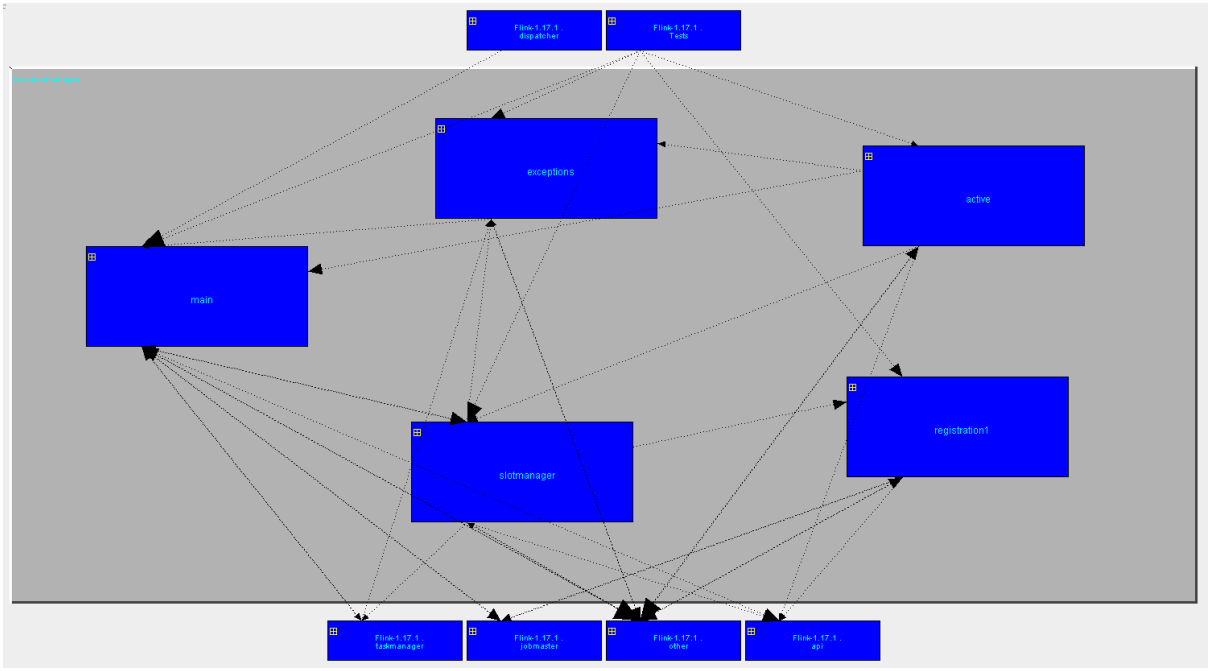

Figure 1.6 - ResourceManager Subsystems and their external dependencies

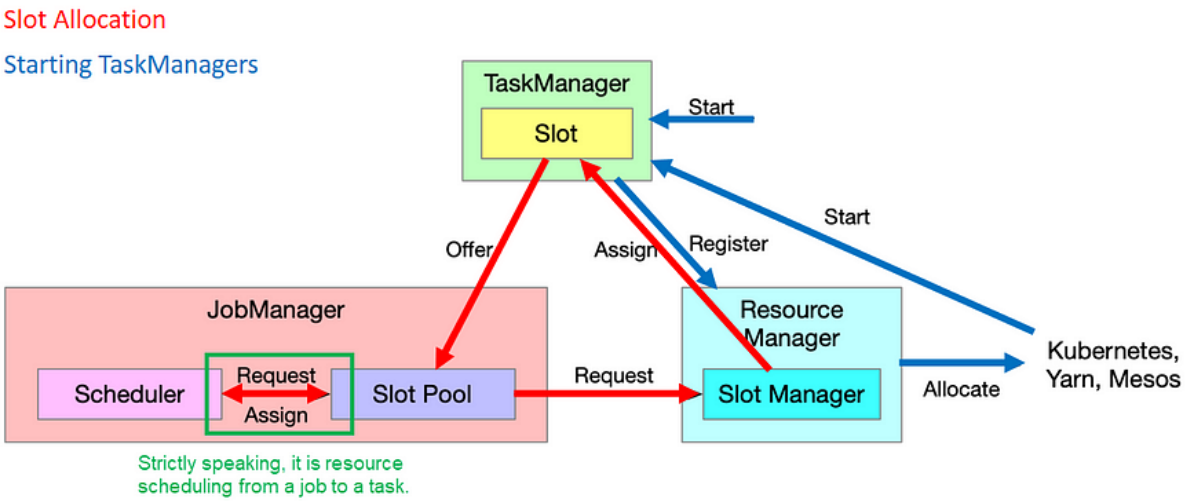

Figure 1.7 - resource allocation and slot assignment by the resource manager

In order for the resource manager to be used, the scheduler within the JobManager will request slots from the slot pool. If the slot pool's resources are sufficient, the slot pool will directly give the slots to the scheduler to use for the overall JobManager. If the slot pool has insufficient resources, the slot pool will then send a request to the slot manager. This can also be seen as a Job requesting resources from the cluster.
What happens from there is that the slot manager, within the resource manager, checks to see if the cluster has sufficient resources to supply the job manager. If so, the resource manager will send an assign command to the task manager, which will then relinquish it's slot to the slot pools.

However, there is also the possibility that there are not enough task managers to supply the slot pool if requested. If the resource deployment mode is active with active resource managers, the active resource manager will help assign additional slots. If the slot manager is expected to send additional resources to a job, and the cluster does not have enough resources, a request will be sent to the underlying resource management subsystem. This will eventually create some task managers that contain task slots that will then be offered to the slot pool, when necessary.

Use Cases

Native Flink on Kubernetes for High-Availability

[2]A Flink cluster normally has one JobManager tracking Jobstatus and coordinating tasks. This makes Flink clusters vulnerable to single point of failure. If the JobManager fails, no more programs can run. A high-availability deployment prevents this by having multiple standby JobManagers to take over as leader when one fails. This can be implemented using Kubernetes which supplies pods for JobManagers as well as TaskManagers.

[2] "Overview | Apache Flink." https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/deployment/ha/overview/. Accessed 3 Nov. 2023.

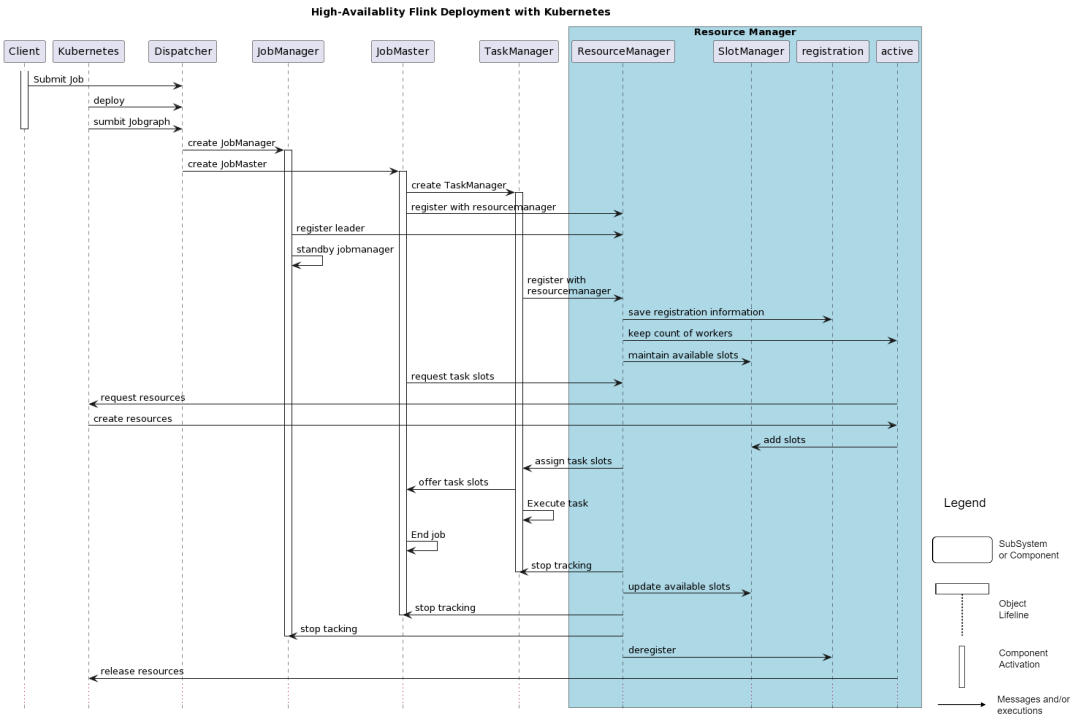

Figure 1.8 - Sequence Diagram showing Flink's top level subsystems and resource manager subsystems interactions

Flink's client has an embedded kubernetes client, which is enough to create the cluster. The sequence of events that takes place shows the resource manager and its subsystem interaction as follows.

[3]Flink's client makes a call to the kubernetes api which submits the job to the dispatcher and creates the JobManager and JobMaster. The resource manager registers these workers as well as any active TaskManagers and keeps track of their heartbeats. The associated information and job artefacts of these workers are stored in the 'register' subsystem and a count stored in the 'active' subsystem. A leader JobManager also registers itself for the high - availability aspect.

When JobMaster requests for task slots, the resource manager will assign those slots to a task manager from available slots maintained by the 'SlotManager' subsystem. If it needs more resources, 'active' is responsible for requesting for more resources from high availability resource providers, in this case kubernetes, and stores them in the SlotManager to be assigned to a TaskManager. The task manager will then offer the slots to the JobMaster to be assigned tasks.

After Job termination, whether it was cancelled, failed or successful, the resource manager will stop tracking the workers and the slots are freed. 'active' will then release the resources.

[3] "How to natively deploy Flink on Kubernetes with High-Availability (HA)." 10 Feb. 2021, https://flink.apache.org/2021/02/10/how-to-natively-deploy-flink-on-kubernetes-with-high-availability-ha/. Accessed 3 Nov. 2023.

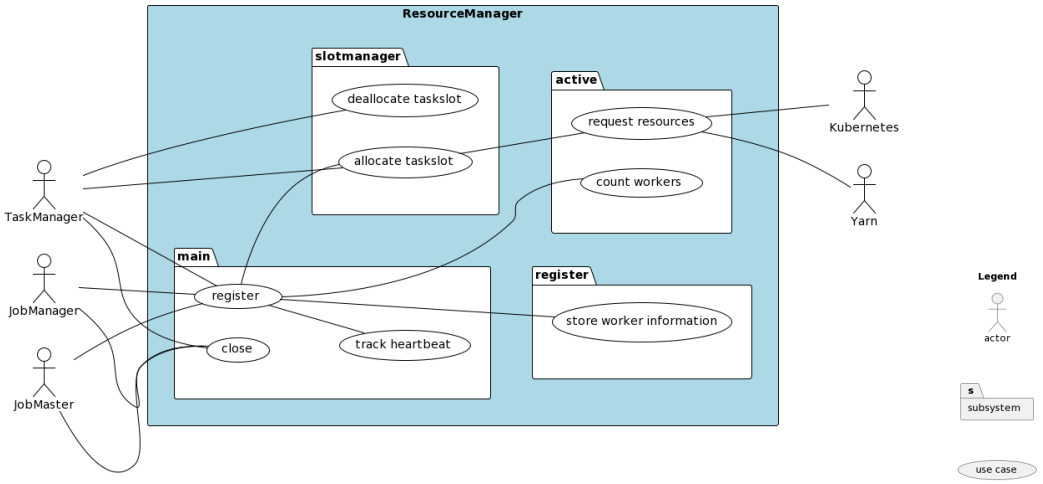

Figure 1.9 - Use Case diagram showing ResourceManager SubSystems in High-Availability deployment using Kubernetes

Lessons Learned

Grok is a language used for manipulating factbases. The team attempted to use the grok tool to simplify the process of determining subsystems within the raw.ta file. However, the team was unsuccessful in this process, which resulted in a lot of time wasted.

Unfamiliarity with the lsedit tool and .contain files is another regret circling this report. Once again, a significant amount of time was spent learning, understanding, and experimenting with .contain files and lsedit capabilities. If the team were to do it differently, they would have certainly learned .contain files and the lsedit tool before beginning the concrete architecture process.

Data Dictionary

- **getsub.py** is a program created by the team to generate a new .contain file from the flink_UnderstandDependencyFile.raw.ta file.
- **Newcontain.py** is a program created by the team to retrieve subsystems from with the created flink_UnderstandDependencyFile.raw.ta file.
- **eClass:** York University's online course website. This site is a hub for students to view and retrieve content posted by their course professors.
- **cLink** is a type of entry in the flink_UnderstandDependencyFile.raw.ta file that signifies a link between two files.
- **$INSTANCE** is an entry in the flink_UnderstandDependencyFile.raw.ta file that signifies the existence of a file.
- **Lsedit** is a tool that can take .ta and .contain files, and generate a graphical representation of the connections between files/subsystems specified in said files.
- **Remote Procedure Call** is a process where a computer program executes a procedure on another system as if the procedure was executed locally.
- **Parallel Processing** is the act of performing multiple tasks simultaneously.
- **Task slot** is a representation of one unit of resource in Apache Flink.
- **Grok** is a language used for manipulating factbases.

Naming Conventions

- **raw.ta** (Used in Derivation Process): Refers to flink_UnderstandDependencyFile.raw.ta
- **.contain** (Used in Lessons Learned): refers to flink_UnderstandDependencyFile.contain
- **RPC** (Used in Architectural Style): Remote Procedure Call

Conclusions

The exploration of the resource manager's architecture within Apache Flink has revealed critical insights into the intricate design choices and structural underpinnings of this pivotal subsystem. Through a meticulous examination of its architectural styles and the application of various design patterns, this study has shed light on the mechanisms employed to manage and allocate resources effectively in distributed computing environments.

Furthermore, this investigation has contributed significantly to the comprehensive coverage of the software system's top-level subsystems. By studying and understanding distinct components such as the resource manager, this collaborative effort aims to offer a holistic view of the architecture and functionalities of Apache Flink.

In conclusion, this report not only serves as an in-depth examination of the resource manager but also stands as a valuable resource for software architects, engineers, and developers seeking to comprehend and refine resource management strategies in complex distributed computing environments. The insights gleaned from this exploration pave the way for more optimized resource utilization and improved performance within Apache Flink and can serve as a foundation for future architectural enhancements in similar distributed systems.

References

https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

https://www.ibm.com/docs/en/aix/7.1?topic=concepts-remote-procedure-callhttps://nightlies.apache.org/flink/flink-docs-master/api/java/org/apache/flink/runtime/resourcemanager/ResourceManager.html

https://jbcodeforce.github.io/flink-studies/architecture/

https://alibaba-cloud.medium.com/data-warehouse-in-depth-interpretation-of-flink-resource-management-mechanism-5c13b531abfa

https://flink.apache.org/2021/02/10/how-to-natively-deploy-flink-on-kubernetes-with-high-availability-ha/

https://issues.apache.org/jira/projects/FLINK/issues/FLINK-32513?filter=allopenissues

EECS 4314 Lecture Slides

EECS 4314 Lab Notes