# Apache Flink

Discrepancy Analysis

**Group Small:**

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

## Abstract

This report offers a comprehensive discrepancy analysis of the conceptual and concrete architecture derived from the Apache Flink framework. The top-level architecture was first explored conceptually and then explored concretely. This study is an in-depth analysis focused on unravelling the underlying differences utilised in the conceptual and concrete architecture. The report describes the reflexion analysis process. The report also compares the concrete architecture to the conceptual architecture of the Apache Flink system. It outlines the interactions between the different subsystems and describes how they work together. The report then delves into the description of the conceptual architecture of the Resource Manager as well as its concrete architecture. It scrutinises the differences between the two. Furthermore, the report gives an explanation of the rationale behind the differences we found in our research. We provide a revised version of a use case presented in previous assignments. 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 optimise resource management strategies in distributed computing environments.

## Introduction

In the first report we discussed the proposed conceptual architecture for the apache flink system. This report described the key features of Flink, the proposed architecture derived from extensive research, and the communication between the subsystems. The second report viewed, discussed, and compared the concrete architecture. This report also introduced the ResourceManager subsystem, which is a system in Apache Flink responsible for resource management and allocation. This report will dive further into the comparison between the conceptual and concrete architecture. It will identify the discrepancies in the top level concrete architecture as well as the ResourceManager subsystem, and provide the discovered rationale behind the discrepancies.

.

## Report Organization

| | |
|---|---|
| **Introduction** | This section will provide the purpose for the report, the report organisation, and salient conclusions. |
| **Conceptual Architecture of Apache Flink** | This section will reiterate the findings of the proposed conceptual architecture. It will discuss the subsystems and their interactions. |
| **Concrete Architecture of Apache Flink** | This section will reiterate the findings of the extracted concrete architecture.  It will discuss the concrete subsystems and their interactions |
| **Reflexion Analysis Process** | This section will describe the process of how the reflexion analysis was performed. |
| **Conceptual vs Concrete Architecture of Apache Flink** | This section will present a reflexion model of the architecture, highlighting the types of discrepancies. This section will also provide the findings of the rationale for the discrepancies. |
| **Conceptual Architecture of Resource Manager** | This section will present and explain the proposed conceptual architecture of the ResourceManager subsystem. It will explain its components and their interactions. |
| **Concrete Architecture of Resource Manager** | This section will present and explain the extracted concrete architecture of the ResourceManager subsystem |
| **Conceptual vs Concrete Architecture of Resource Manager** | This section will compare the conceptual and concrete architecture of the Resource Manager subsystem. And explain any differences. |
| **Rationales Behind Differences** | This section will provide a higher level rationale for the differences between the architecture. |
| **Use Cases** | This section will present modified use case diagrams that conform with the concrete architecture of the resource manager subsystem. |
| **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 summarise all key findings and any proposals for the future. |
| **Lessons Learned** | This section will highlight and explain any lessons learned, and elements that the team wishes they knew beforehand or did differently. |
| **References** | References and Sources. |

# Conceptual Architecture of Apache Flink

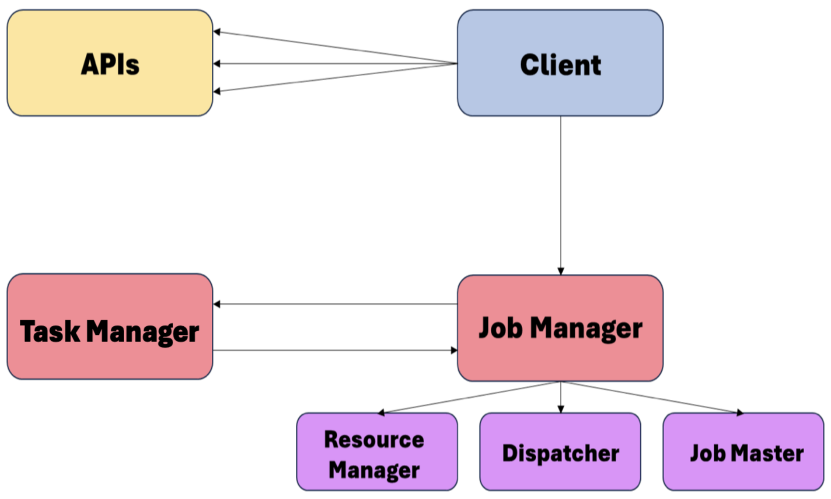


Figure 1.0 conceptual architecture of Apache Flink

Our conceptual architecture above is our way of reasoning and communicating about Apache Flink. More specifically it is how we first envisioned the system's architecture, the different subsystems within Apache Flink and the data flow between them.

- **APIs:** serves as the interface between the client applications and the backend system. It also provides a set of protocols for creating, managing, and monitoring jobs. It facilitates communication with the distributed system through a defined set of commands and functions.
- **Client:** this is the entry point for users or external systems to submit jobs to the system, it is responsible for setting up job-specific configurations and initiating job processing. It may also provide tools for job monitoring and management after submission.
- **Job Manager:** the central control unit that manages the execution of jobs. Handles job scheduling, recovery from failures, and coordination between the other components. It also orchestrates the distribution of the computational tasks to various Task Managers. According to the diagram it assigns tasks to the Task Manager
- **Task Manager:** executes the tasks assigned by the Job Manager. Manages the execution of the tasks on a node in the cluster. Handles the processing of data, maintaining buffers, and executing the operators as defined in the job. Gives the Job Manager an update of the state of the executed task.
- **Resource Manager:** manages the distribution and allocation of computational resources such as memory, CPU, and network bandwidth. Interacts with the cluster infrastructure to provision and deprovision resources as needed. May also deal with resource isolation and ensuring jobs do not exceed their allocated share.
- **Dispatcher:** responsible for accepting job submissions and dispatching them to the Job Manager. May also provide an entry point for job submission when multiple Job Managers are present in a larger cluster. Can act as a load balancer to distribute the workload evenly across the system.
- **Job Master:** a potential sub-component of the Job Manager responsible for managing the lifecycle of a specific job. Handles the specifics of job execution, like task deployment, checkpointing, and scaling. Acts as a mediator between the Job Manager and Task Managers for job-specific communications.

# Concrete Architecture of Apache Flink

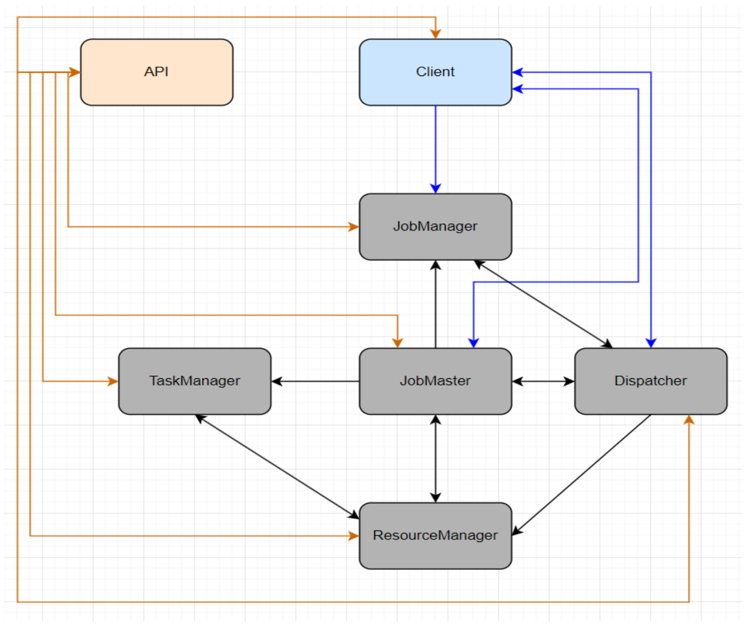
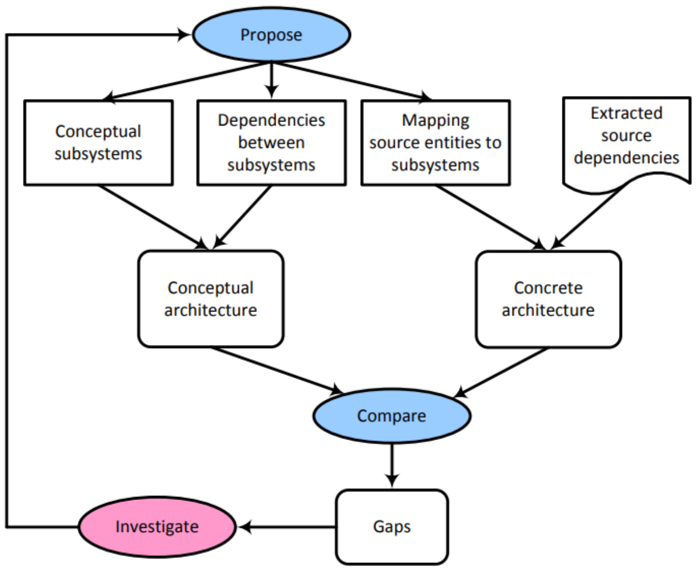


Figure 1.1 concrete architecture of Apache Flink

The image illustrates a more detailed architecture of a distributed system, such as Apache Flink, with various components connected through directional arrows indicating the flow of operations and communications. Here's a description of the architecture:

- **API:** this is the interface for defining jobs and data transformations. It is likely used by developers to program the data processing logic that Flink will execute. It enables every subsystem to utilise the libraries directly
- **Client:** the client submits the data processing job to Flink. It communicates with the JobManager to submit the serialised job graph for execution. The dispatcher sends the response directly to the client
- **JobManager:** this is the central coordinator of the Flink Cluster. It is responsible for the scheduling and coordination of jobs and tasks. The JobManager receives the job from the client, prepares it for execution, and controls its execution.
- **JobMaster:** it might be a part of the JobManager or a dedicated component responsible for managing a single job. Manages the execution of a job and coordinates between TaskManagers. Handles task execution, checkpointing, recovery, and updates to the job status.
- **Dispatcher:** it is responsible for job reception and forwarding to the JobManager. It may also act as a gateway for the REST API to allow job submissions and provide cluster information. Depends on the Resource Manager to provide the resources necessary to generate a response for the client.
- **TaskManager:** these are the worker nodes that execute the tasks assigned to them by the JobManager. They run the actual tasks, process the data, and communicate the results or updates back to the JobManager.
- **ResourceManager:** it manages the allocation and deallocation of resources for TaskManagers. Ensures efficient distribution of resources and scaling of the cluster according to the workload.

## Reflexion Analysis Process


Propose
Conceptual subsystems
Dependencies between subsystems
Mapping source entities to subsystems
Extracted source dependencies
Conceptual architecture
Concrete architecture
Compare
Gaps
Investigate

[3]

Figure 1.3 The reflexion analysis process

Above is the reflexion analysis process. The proposal, intermediate steps, and generation of the conceptual architecture was completed during the first report. The concrete architecture was extracted during the second report. This report compares the two architectures and investigates gaps between the two. This process of discovering and assessing gaps between architectures will be referred to as the discrepancy analysis. The discrepancy analysis began with identifying gaps between the architectures. This report will focus on two gaps. Absences are dependencies that were presented in the concrete architecture, but are not present in the extracted concrete architecture. Divergences are dependencies that are not  in the conceptual architecture, but were introduced  in the extracted concrete architecture. Once these gaps were identified, and the responsible subsystems were identified, the flink_UnderstandDependencyFile.raw.ta file was examined to dig deeper and find the exact class or classes responsible for the discrepancy. These classes were accessed and the comments in the code were examined to find insight on why this dependency was added/removed. Reading the comments also provided a better understanding of the purpose of the class and how it connects to the class in the other subsystem. From here the Apache Flink GitHub repo and JIRA  issue tracker were viewed to find desired information about the introduction to, or removal of the dependency

## Conceptual vs Concrete Architecture of Apache Flink

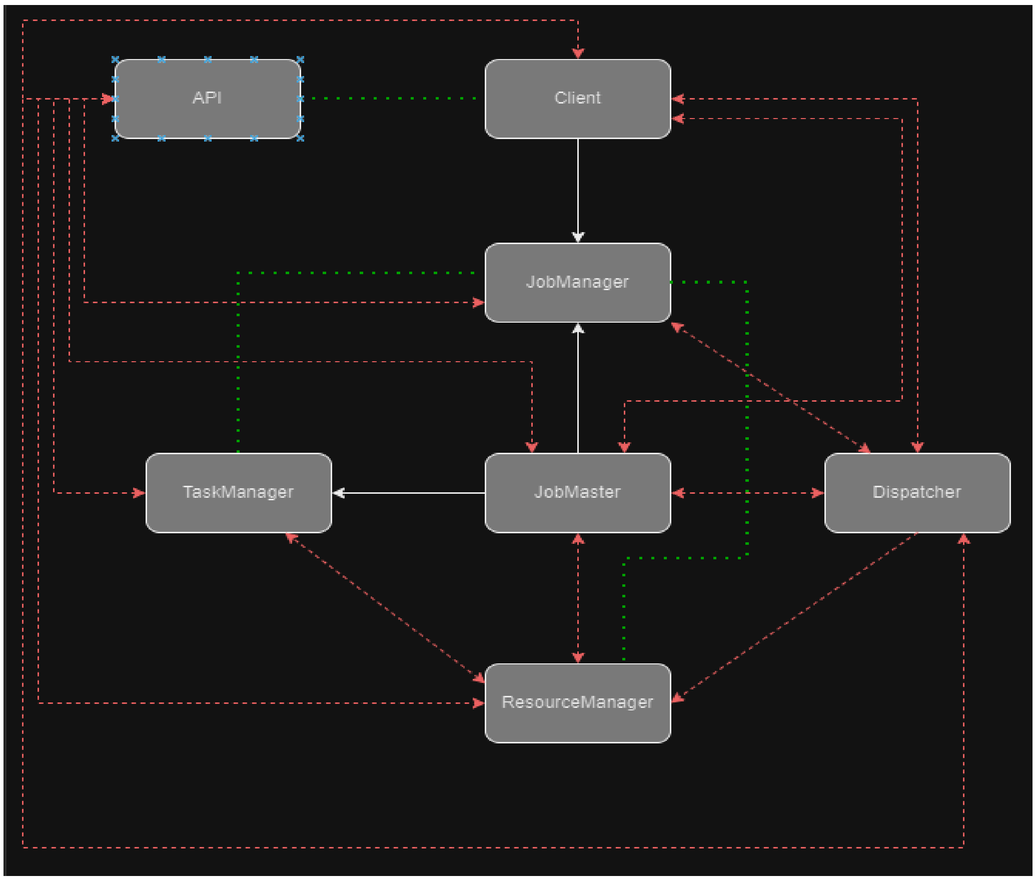


Figure 1.4 The discrepancies discovered in the reflexion analysis process

**Divergence(- - - - -)**
Dependencies present in the concrete architecture, but missing in the conceptual architecture.

**Absence(• • • • • • •)**
Dependencies present in the conceptual architecture, and missing in the concrete architecture

Above is a diagram highlighting all discrepancies between the conceptual and concrete architecture. Upon investigating the discrepancies. Several divergences were discovered to be misunderstandings that the team had when developing the conceptual architecture. In the conceptual architecture, it was believed that only the client communicates with the API. However, the concrete architecture reveals that all subsystems communicate with the API. Similarly, it was believed that the ResourceManager, JobMaster, and Dispatcher subsystems only communicate to the JobManager. This again was proven to be false as the subsystems are interconnected with each other. Below are the findings of these dependencies and the rationales behind them.

| Divergence (ResourceManager ⟷ JobMaster) | |
|---|---|
| WHAT | connectToResourceManager (flink-runtime/…/runtime/jobmaster/slotpool/SlotPool.java) depends on ResourceManagerGateway (flink-runtime/…/runtime/resourcemanager/ResourceManagerGateway.java) |
| WHO | Stephen Ewen (Senior Programmer, Co-Creator) |

|  |  |
|---|---|
| WHEN | Created 08/02/2017 Resolved 08/29/2017 |
| WHY | The following are core aspects of the ResourceManager design:<br>The ResourceManager no longer has a resource pool size, but receives individual requests for slots. That way, jobs can request TaskManagers of different resources (Memory/CPU).<br><br>/**<br>* Connects the SlotPool to the given ResourceManager. After this method is called, the SlotPool<br>* will be able to request resources from the given ResourceManager.<br>*<br>* @param resourceManagerGateway The RPC gateway for the resource manager. |

| Divergence (JobMaster ←→ Dispatcher) |  |
|---|---|
| WHAT | (flink-runtime/…/runtime/jobmaster/) two way dependency on (flink-runtime/…/runtime/dispatcher/) |
| WHO | Till Rohrmann (Flink PMC Member) |
| WHEN | Created 11/Feb/18 00:37 Resolved 23/Feb/18 09:25 |
| WHY | In order to call the JobMaster#rescaleJob via Rest handlers, it has to be exposed via the Dispatcher.<br><br>Extracted from jobResult.java<br>/**<br>* Similar to {@link org.apache.flink.api.common.JobExecutionResult} but with an optional {@link<br>* SerializedThrowable} when the job failed.<br>*<br>* &lt;p&gt;This is used by the {@link JobMaster} to send the results to the {@link Dispatcher}. |

| Divergence (Dispatcher ←→ ResourceManager) |  |
|---|---|
| WHAT | flink-runtime/…/dispatcher/Dispatcher.java to flink-runtime/…/runtime/resourcemanager/ResourceOvervi |
| WHO | Till Rohrmann (Flink PMC Member) |
| WHEN | Created Oct 10, 2017 Resolved Oct 12, 2017 |

| | |
|---|---|
| WHY | This commit implements the ClusterOverview generation on the Dispatcher. In order to do this, the Dispatcher requests the ResourceOverview from the ResourceManager and the job status from all JobMasters. After receiving all information, it is compiled into the ClusterOverview. |

| Divergence (ResourceManager ←→ TaskManager) | |
|---|---|
| WHAT | flink-runtime/…/runtime/resourcemanager/slotmanager/SlotManager.java<br>flink-runtime/…/runtime/rest/messages/taskmanager/SlotInfo.java |
| WHO | YangZe Guo (Flink Committer) |
| WHEN | Created 15/Mar/21 Resolved 24/Mar/21 |
| WHY | It would be helpful to allow retrieving detailed information of slots via rest api.<br><br>JobID that the slot is assigned to<br>Slot resources (for dynamic slot allocation)<br><br>Such information should be displayed on webui, once fine-grained resource management is enabled in future. |

| Divergence (Dispatcher ←→ JobManager) | |
|---|---|
| WHAT | flink-runtime/…/runtime/Dispatcher/Dispatcher.java two way with<br>flink-runtime/…/runtime/jobmanager/SubmittedJobGraph.java |
| WHO | Till Rohrmann and Chesnay Schepler (Flink PMC Members) |
| WHEN | Created 04/Jul/17 16:04 Resolved 11/Jul/17 17:47 |

| | |
|---|---|
| WHY | The Dispatcher is responsible for receiving job submissions, persisting the JobGraphs, spawning JobManager to execute the jobs and recovering the jobs in case of a master failure. This commit adds the basic skeleton including the RPC call for job submission.<br><br>Add cleanup logic for finished jobs<br><br>Pass BlobService to JobManagerRunner |

| Divergence (Client ←→ Dispatcher) | |
|---|---|
| WHAT | flink-1.17.1/flink-clients/…/client/deployment/application/ApplicationClusterEntryPoint.java to flink-1.17.1/flink-runtime/…/runtime/dispatcher/ExecutionGraphInfoStore.java |
| WHO | Till Rohrmann (Flink PMC Member) |
| WHEN | Created 03/Jul/17 22:40 Resolved 26/Jul/17 23:25 |
| WHY | Implement a generic entry point for Flink sessions. This ClusterEntryPoint has to start a ResourceManager, the Dispatcher component and the cluster's RESTful endpoint. This class could serve as the basis for a Mesos- and YarnEntryPoint to run Flink sessions.<br><br>Maybe we can use a common base for the session and the per-job mode. The session has to start a dispatcher component and the per-job mode retrieves the JobGraph and directly starts a JobManager with this job. |

| Divergence (Client ←→ JobMaster) | |
|---|---|
| WHAT | flink-1.17.1/flink-clients/…/client/program/rest/RestClusterClient.java to flink-1.17.1/flink-runtime/…/runtime/jobmaster/JobResult.java |
| WHO | Till Rohrmann (Flink PMC Member) |
| WHEN | Created 03/Jul/17 21:33 Resolved 26/Aug/17 09:13 |

In order to communicate with the cluster from the RESTful client, we have to implement a RESTful cluster endpoint. The endpoint shall support the following operations:

- List jobs (GET): Get list of all running jobs on the cluster
- Submit job (POST): Submit a job to the cluster (only supported in session mode)
- Get job status (GET): Get the status of an executed job (and maybe the JobExecutionResult)
- Lookup job leader (GET): Gets the JM leader for the given job

This endpoint will run in session mode alongside the dispatcher/session runner and forward calls to this component which maintains a view on all currently executed jobs.

In the per-job mode, the endpoint will return only the single running job and the address of the JobManager alongside which it is running. Furthermore, it won't accept job submissions.

## Conceptual Architecture of Resource Manager

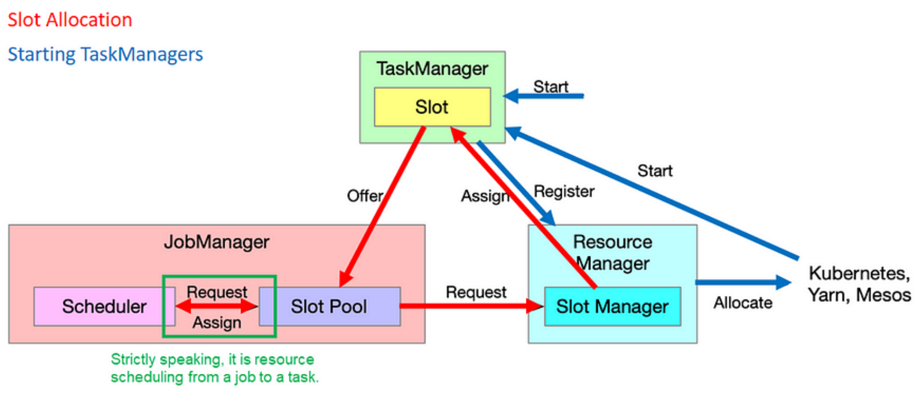


Figure 1.5 conceptual architecture of the resource manager

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 its 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.

## Concrete Architecture of Resource Manager

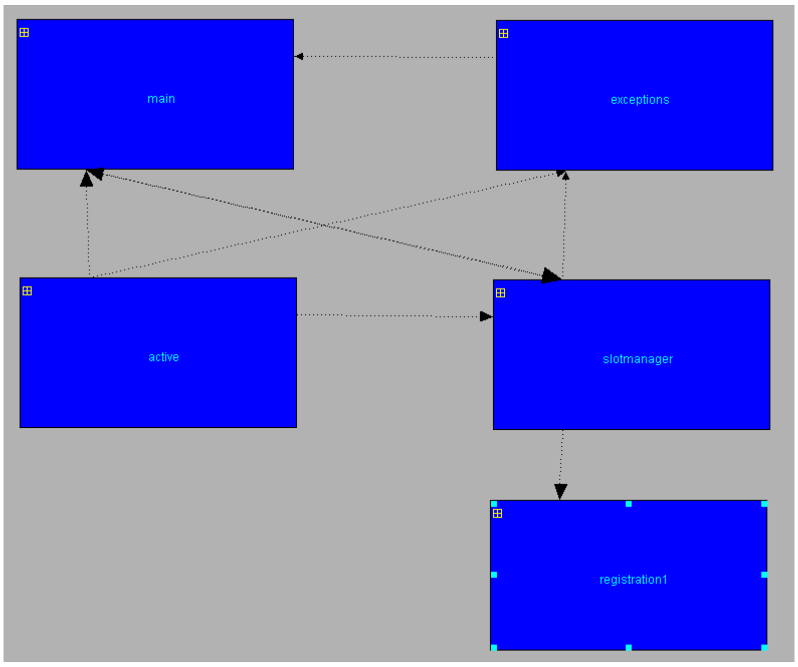


Figure 1.6  concrete architecture of the resource manager

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.

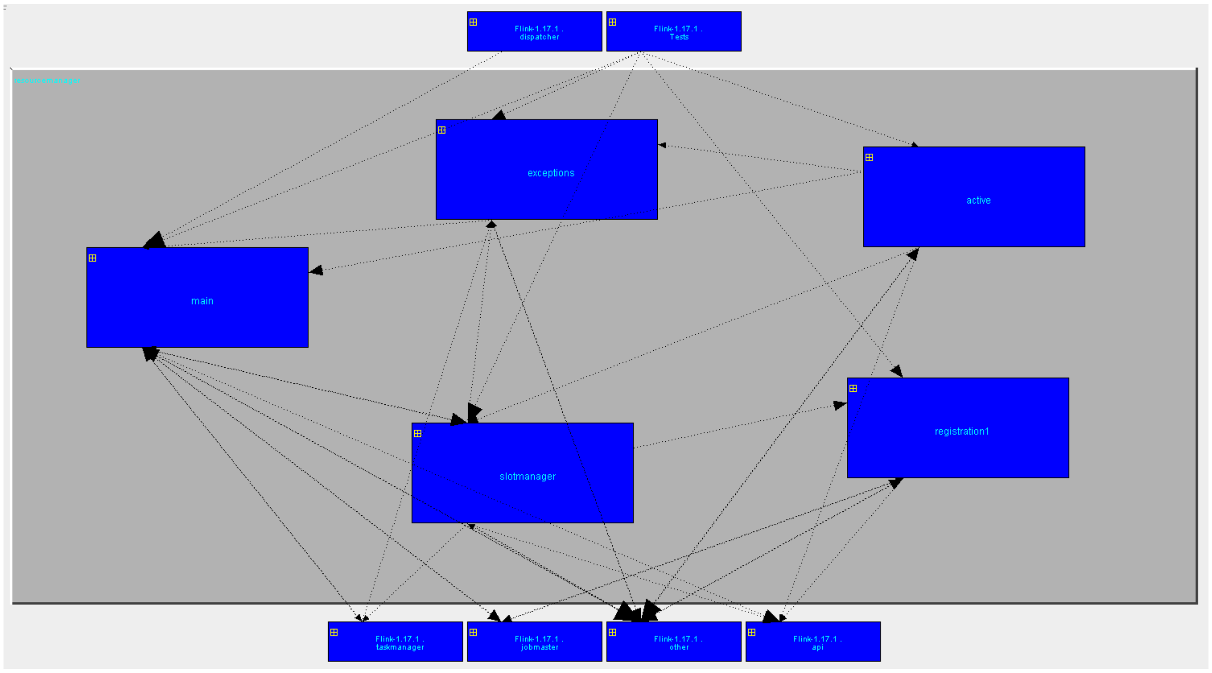


Figure 1.7  Interactions between the resource manager subsystems and Flink’s top level subsystems

## Conceptual vs Concrete Architecture of Resource Manager

The conceptual architecture versus the concrete architecture is very different. The conceptual architecture made no reference to any of the other subsystems within the resource manager other than the slot manager. It puts more of the emphasis on the surrounding subsystems and it takes everything through simple requests and assigns commands. On top of that as well, there was no explanation to how the slot pool was known, and whether or not they were registered, free or pending. The concrete architecture actually delves much deeper into the subsystem, and its own subsystems within. It covers the main section and the slot manager, but also displays other sections such as the exceptions, registrations and active sections. It elaborates further than the conceptual architecture as well in terms of what interacts with what top level subsystem.

## Rationales Behind Differences

Concrete Architecture: concerned with the actual implementation details, technologies, and specific components of a system. It deals with tangible elements such as databases, servers, APIs, programming languages, and hardware configurations. Architects working on concrete architecture need to consider real-world constraints, performance optimizations, and integration points. Concrete architecture is developed later in the software development process, typically during the implementation or construction phase. It is used by developers, engineers, and other technical stakeholders to guide the actual building of the system. The emphasis is on practicality, feasibility, and meeting specific technical requirements. Concrete architecture tends to be less flexible because it is closely tied to specific technologies and platforms. Changes to the concrete architecture may require significant modifications to the codebase. Communicates detailed technical specifications and configurations to developers and technical teams. Helps ensure a common understanding among the implementation team. Subject to evolution and iteration as the system is developed and more details become available. It evolves based on practical considerations, feedback from implementation, and changing requirements.

Conceptual Architecture: operates at a higher level of abstraction. It focuses on the overarching design principles, key concepts, and the structure of the system without getting into specific implementation details. Architects working on conceptual architecture are more concerned with defining the high-level vision, requirements, and functionality of the system. Conceptual architecture is created early in the software development life cycle, often during the planning or initiation phase. It is used to communicate the broad design goals and ideas to non-technical stakeholders, such as project managers, business analysts, and executives. The focus is on aligning the system design with business objectives, user needs, and strategic goals. Conceptual architecture is more adaptable and open to changes. It provides a foundation for making informed decisions about technology choices and system structure without committing to specific implementations. Communicates the essence of the system's design to non-technical stakeholders. Enables alignment of the project with business objectives and facilitates decision-making at a strategic level. Provides a stable foundation that guides the evolution of concrete architecture. Changes to the conceptual architecture are typically driven by shifts in business strategy or high-level project goals.

# Use Cases

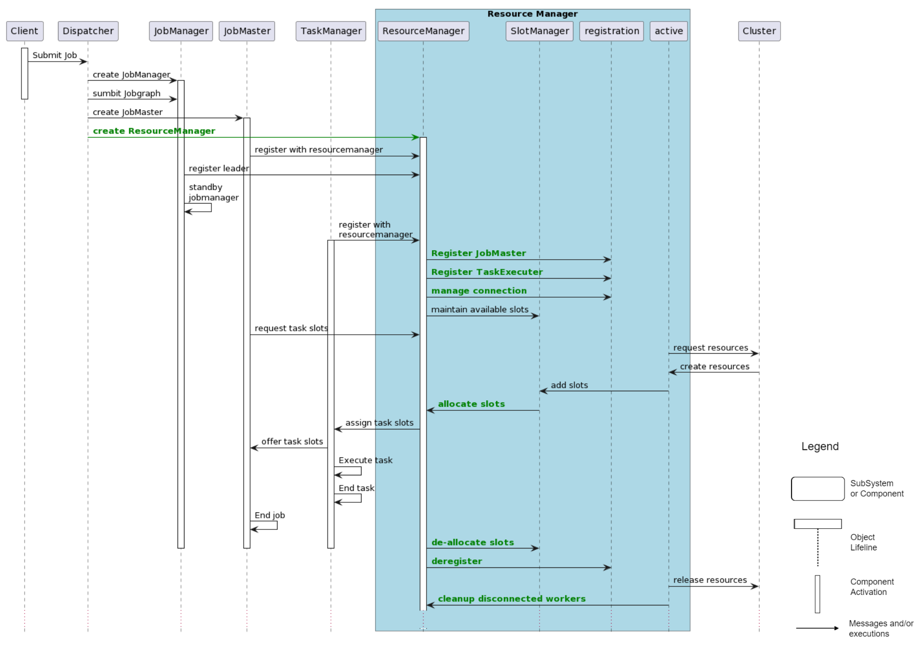


Figure 1.8 Updated sequence diagram for general Flink application cluster

In the concrete architecture analysis we used a sequence diagram to show high-availability Flink deployment using Kubernetes to showcase interactions between Flink's top level subsystems as well as the resource manager's subsystems. Here we have generalised that use case to include any cluster framework used and have updated some of the interactions to reflect the discovered dependencies and highlight more details.

The main change is the addition of "create resource manager" by the dispatcher[1]. "Register JobMaster", "Register TaskExecuter" and "manage connections" replaced "save registration information" as the registration subsystem had separate implementations for each. "allocate slots" and "deallocate slots" were added to show the slot manager's role in managing task slots. And finally after the termination or disconnection of workers i.e. TaskManager, JobMaster or JobManager the main resource manager and active resource manager "deregister" and "cleanup disconnected workers".

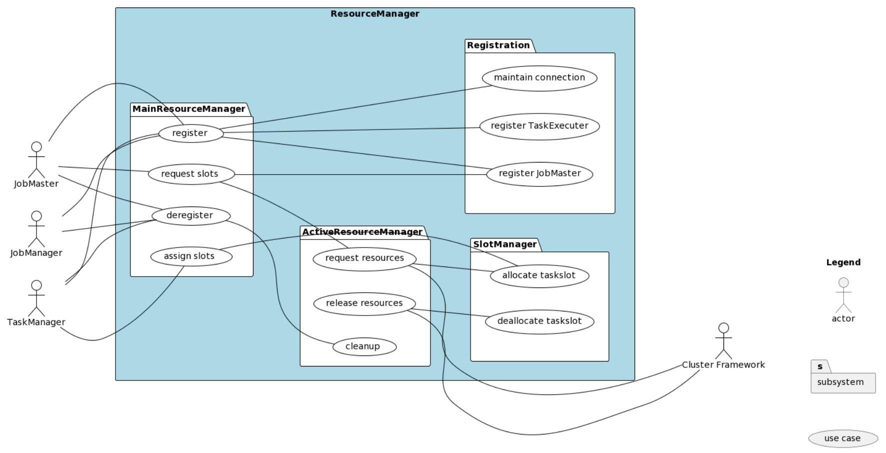


Figure 1.9 Updated use case diagram of Resource management subsystems

The use case diagram shows the changes mentioned above and reflects how the resource manager components are implemented. One thing to note here is that request for slots are processed through the "main" resource manager and that the slot manager does not directly allocate slots to the task manager[2].

## Data Dictionary

**Conceptual Architecture:** This is the proposed architecture that was derived from research

**Concrete Architecture:** This is the concrete architecture extracted from a visualizer

**flink_UnderstandDependencyFile.raw.ta:** This refers to the dependency file in the concrete architecture report. This file contains lists all classes and connections between classes in the Flink source code folder.

**GitHub:** A code repository that is home to apache flinks source code. This repository stores all changes made to the source code folder, tracking which user and when they made the changes.

**JIRA:** An bug/issue tracking system. This software is used to organize pending bug fixes, improvement, issues, upgrades, features, etc. This software also tracks which users and when the ticket request was made and resolved.

**Divergence:** Dependencies present in the concrete architecture, but missing in the conceptual architecture.

**Absence:** Dependencies present in the conceptual architecture, and missing in the concrete architecture

**PMC:** Apache Attic Committee, the official controlling body of the Apache Flink project.

## Naming Conventions

**PascalCase:** Used when naming Apache Flink components or subsystems. It is also used for Flink file naming. This is the default case used in Apache Flink documentation.

## Lessons Learned

**1.Github Version History**

- *What we learned:* GitHub's version history is a powerful tool for tracking changes, but it might not always provide a straightforward way to view when specific files were added.
- *Lesson:* Understanding the limitations of the default version history can help us explore additional strategies for obtaining the information we need.

**2. Viewing When Files were Added**

- *What we learned:* The ability to easily view when files were added is crucial for understanding the project's evolution.

- *Lesson:* While GitHub provides some file history information, there might be cases where a more specialised approach is needed to get a comprehensive view of file additions.

**3. File History Challenges**

- *What we learned:* GitHub's file history feature might not always be sufficient for our needs, especially when dealing with complex projects or specific file tracking requirements.
- *Lesson:* Exploring alternative methods or tools for tracking file history can provide a more detailed and accurate representation of changes over time.

**4. Using Scripts for Automation**

- *What we learned:* Developing and using scripts, such as a Python script for extracting dependencies, can significantly enhance efficiency and automate repetitive tasks.
- *Lesson:* Leveraging scripts is a valuable practice for tasks like dependency extraction, saving time and reducing the likelihood of human error.

**5. Python Script for Extracting Dependencies**

- *What we learned:* The Python script for extracting dependencies proved to be a useful tool, streamlining a manual process and improving accuracy.
- *Lesson:* Investing in script development aligned with project needs can contribute to overall project efficiency and maintainability.

**6. Efficiency Consideration**

- *What we learned:* Efficiency is a critical factor in managing projects, and scripts can play a crucial role in achieving automation and speed.
- *Lesson:* Continuously evaluating and optimising our workflows, especially through automation, contributes to overall project efficiency.

## References

[1]https://github.com/apache/flink/blob/master/flink-runtime/src/main/java/org/apache/flink/runtime/dispatcher/DispatcherServices.java#L130
[2]https://github.com/apache/flink/blob/master/flink-runtime/src/main/java/org/apache/flink/runtime/resourcemanager/ResourceManager.java#L716
https://github.com/apache/flink
https://issues.apache.org/jira/projects/FLINK/summary
Software Reflexion Models: Bridging the Gap between Source and High-Level Models
[3] EECS4314 Lecture Slides