Санкт–Петербургский государственный университет

РОМАНЫЧЕВ Леонид Романович

Выпускная квалификационная работа

Проектирование агентского приложения для специализированной распределенной вычислительной системы SPD Online filter

Уровень образования: магистратура
Направление 02.04.02 «Фундаментальная информатика и информационные технологии»
Основная образовательная программа «Распределенные вычислительные технологии» №21/5827/1

Научный руководитель:
Кафедра компьютерного моделирования
и многопроцессорных систем,
д.т.н. Дегтярев Александр Борисович

Руководитель от организации:
ведущий программист, МЛИТ, ОИЯИ,
к.т.н. Олейник Данила Анатольевич

Рецензент:
заместитель начальника научно-экспериментального
отдела встречных пучков, ЛЯП, ОИЯИ,
к.ф. – м.н. Жемчугов Алексей Сергеевич

Санкт-Петербург
2023

Saint–Petersburg State University

ROMANYCHEV Leonid Romanovich

Graduation project

Designing a data processing process management service on a specialized distributed computing system SPD Online filter

Level of education: Master level
Main field of study 02.04.02 «Fundamental Informatics and Information Technology»
Academic programme title «Distributed Computational Technologies»
No21/5827/1

Scientific supervisor:
Dr. in Technical science
A.B.Degtyarev

Supervisor from organisation:
lead software developer, MLIT, JINR
PhD in Technical science
D.A.Oleynik

Reviewer:
Deputy Head of the Scientific and Experimental
Department of Oncoming Beams, LNP, JINR,
PhD in Physical and Mathematical sciences
A.S.Zhemchugov

Saint–Petersburg
2023

Оглавление

Список сокращений и условных обозначений **3**
Введение **5**
Постановка задачи **10**
Глава 1. Анализ существующих решений **11**
1.1 PanDA Pilot 12
1.2 DIRAC Pilot 23
1.3 RADICAL-Pilot 26
Глава 2. Общая архитектура **30**
2.1 Workload Management System (WMS) 30
2.2 Формирование задач 32
2.3 Жизненный цикл задач (Jobs) 32
2.4 Взаимодействие WMS и Pilot 33
2.5 Job-executor 33
Глава 3. Архитектура VISOR Pilot **34**
Глава 4. Разработка агентского приложения **37**
4.1 Выбор стека технологий 37
4.2 Интерфейс взаимодействия с JobExecutor 38
4.3 RabbitMQ 39
4.4 Принимаемые значения статуса задачи 39
4.5 Структура проекта 40
Выводы **42**
Заключение **43**
Список источников **44**
Приложения **49**

Список сокращений и условных обозначений

Термины	**Определения**
ОИЯИ	Объединенный Институт Ядерных Исследований
МЛИТ	Лабораторией Информационных Технологий имени Мещерякова
NICA	Nuclotron based Ion Collider fAcility
SPD	Spin Physics Detector
БАК (LHC)	Большой Адронный Коллайдер (Large Hadron Collider)
ATLAS	A Toroidal LHC Apparatus
LCG	LHC Computing Grid
БД	База данных
GRID	Является географически распределённой инфраструктурой, объединяющей множество ресурсов разных типов, доступ к которым пользователь может получить из любой точки, независимо от места их расположения.
Задание (task)	Представляет собой единицу рабочей нагрузки по обработке блока данных
Задача (job)	Единица рабочей нагрузки для обработки элемента (или нескольких элементов) блока данных
JSON	JavaScript Object Notation. Текстовый формат обмена данными, основанный на JavaScript
HTTP	HyperText Transfer Protocol. Протокол прикладного уровня передачи данных, изначально — в виде гипертекстовых документов в формате HTML, в настоящее время используется для передачи произвольных данных.
AMQP	Advanced Message Queuing Protocol. Протокол прикладного уровня для передачи сообщений между компонентами системы.
API	Application Programming Interface. Описание способов взаимодействия одной компьютерной программы с

	другими.
REST	Representational State Transfer. Архитектурный стиль / набор правил взаимодействия компонентов распределённого приложения в сети.
URL	Uniform Resource Locator
Git	Распределённая система управления версиями.

Введение

SPD (Spin Physics Detector) [1] это одна из экспериментальных установок входящих в мегасайнс проект NICA (Nuclotron-based Ion Collider fAcility) [2], которая строится в ОИЯИ (г. Дубна, Россия) [3]. Схема комплекса представлена на рисунке 1. Основная цель проводимых на установке исследований – проверка основ квантовой хромодинамики (КХД), фундаментальной теории сильных ядерных взаимодействий, путем изучения поляризованной структуры нуклона и спиновых явлений при столкновении продольно и поперечно поляризованных протонов и дейтронов с энергией центра масс до 27 ГэВ. и светимостью до 10^{32} см$^{-2}$ с$^{-1}$.

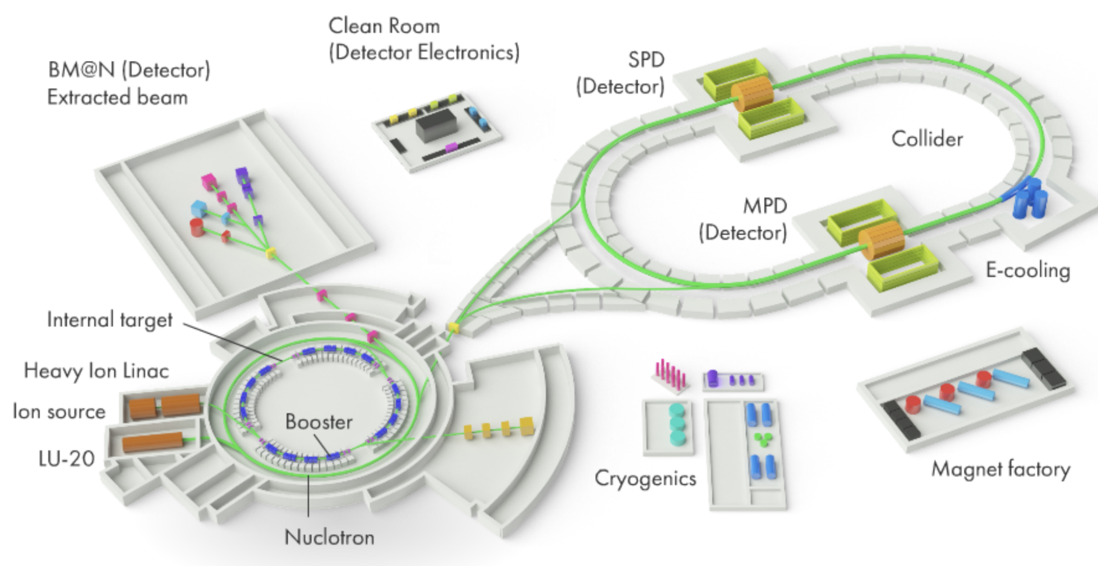

Рис. 1. Схема комплекса NICA.

Детектор SPD (рис. 2) – это универсальный 4π-спектрометр, основанный на современных технологиях [4]. Общее количество каналов регистрации в установке SPD составляет около 500000. С учетом ожидаемой максимальной частоты пересечения пучков около 12 МГц и ожидаемой частоты возникновения интересующих эксперимент событий около 3 МГц, суммарный поток данных с детектора можно оценить как 20 ГБ / с, что

эквивалентно 200 ПБ/год (для эксперимента предполагается выделить 30% от времени работы коллайдера). Сбор, обработка и хранение такого объема данных не представляется экономически возможным.

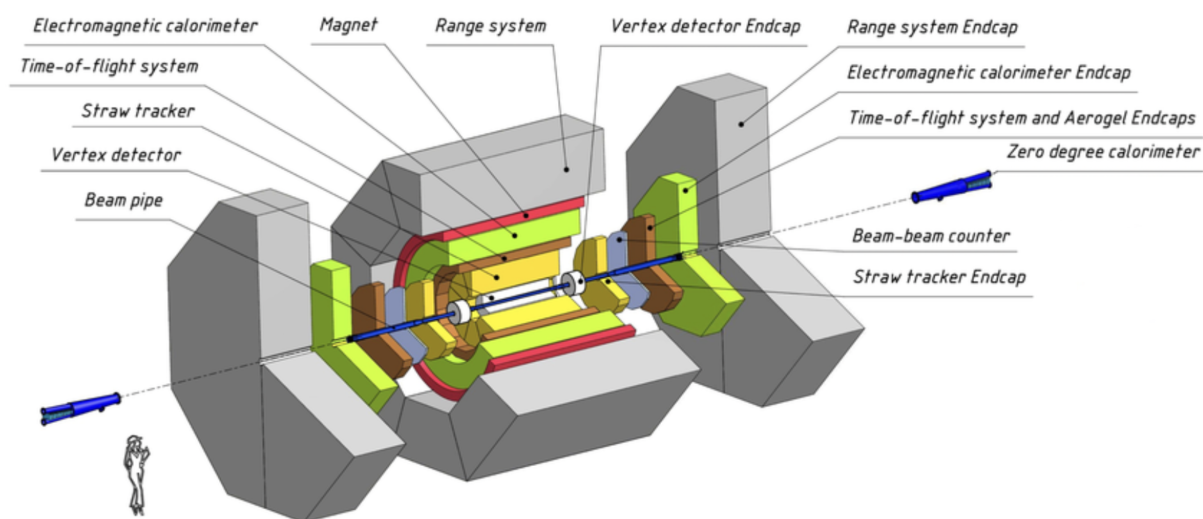

Рис. 2. Экспериментальная установка SPD.

Эксперименты, проводимые на ускорителях, предоставляют возможность исследования взаимодействий ускоренных частиц между собой или с веществом. Такие взаимодействия называются «событием» (см. рис. 3). Каждое событие не зависит от другого и поэтому может обрабатываться независимо. В обработке данных ФВЭ (Физики Высоких Энергий) событие является наименьшей единицей.

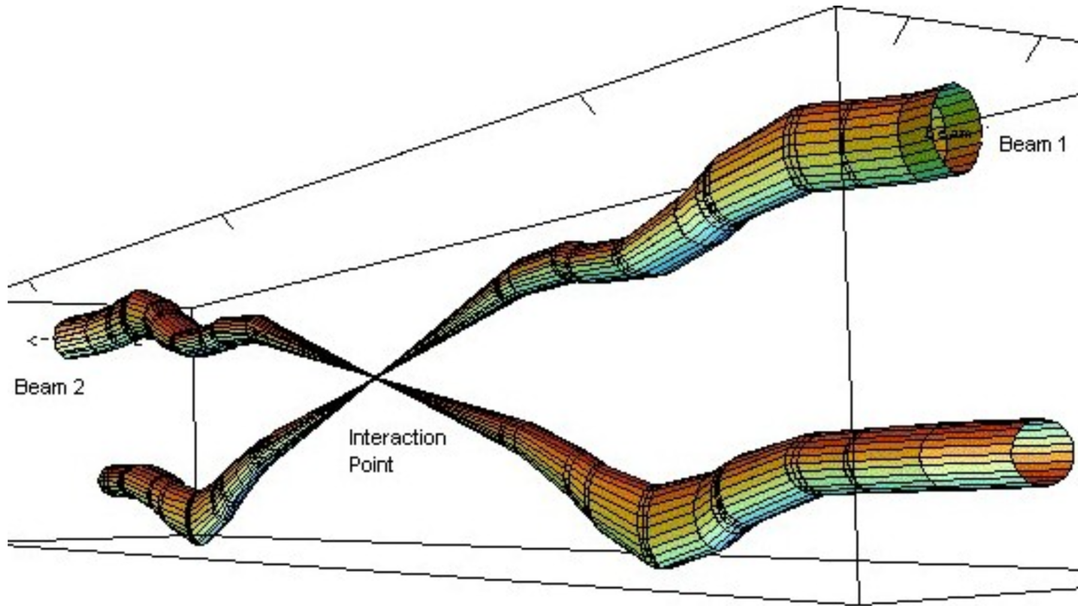

Рис. 3. Иллюстрация «события» в коллайдере.

Большой поток данных образуется в результате использования безтриггерной системы. Он изображен на рисунке 4.

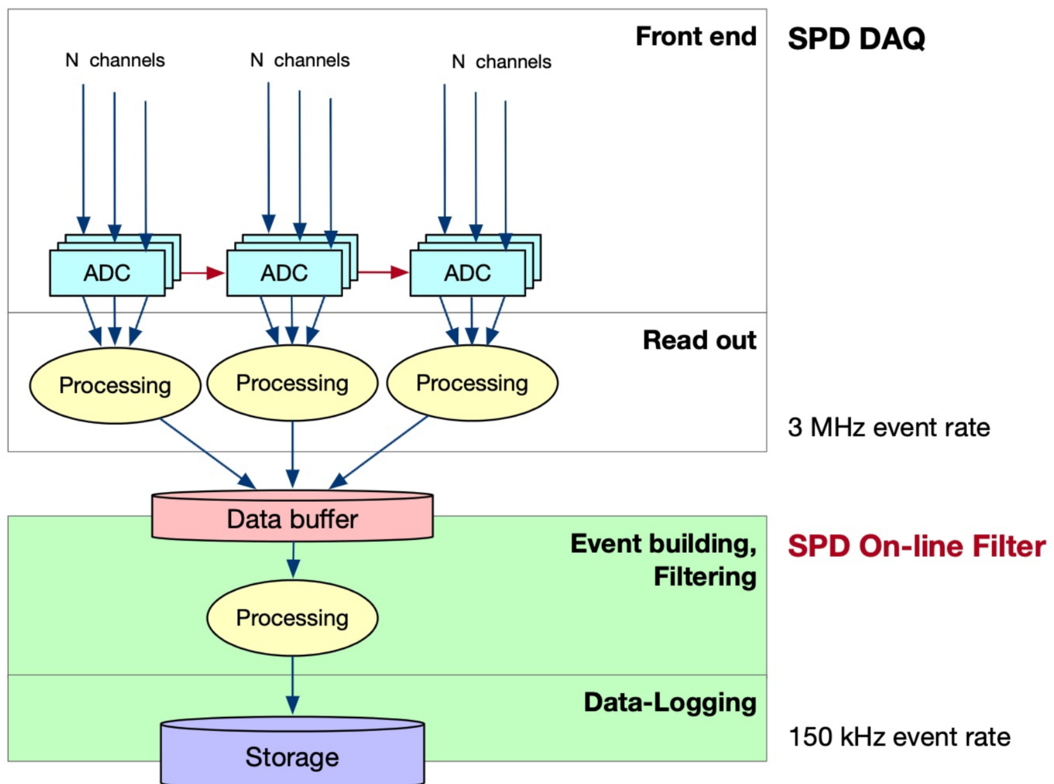

Рис. 4. Схема «triggerless» DAQ.

На выходе из DAQ получается наборы сигналов с сенсоров, организованных во временные блоки, превосходящие по времени период между событиями (в одном таком блоке предполагается несколько событий). В случае с триггерной системой, эти сигналы сразу же ассоциированы с каким либо событием, а в случае с безтриггерной системой из полученного набора агрегированных с разных сенсоров сигналов нужно сначала выявить события, а затем уже отобрать нужные. Безтриггерный DAQ используется для того, чтобы минимизировать систематическую ошибку при изучении микроскопических эффектов. Количество файлов на выходе из DAQ в безтриггерной системе существенно возрастает. Эти файлы можно обрабатывать независимо. В связи с этим необходима специализированная высокопропускная вычислительная система, которая бы обеспечивала:

- выявление наборов сигналов ассоциированных с событиями,
- из числа выявленных событий отбор только тех, которые необходимы в рамках текущего физического исследования,
- подготовка данных для систем мониторинга данных и прочей первичной обработки.

За все эти процессы отвечает вычислительная установка SPD OnLine Filter.

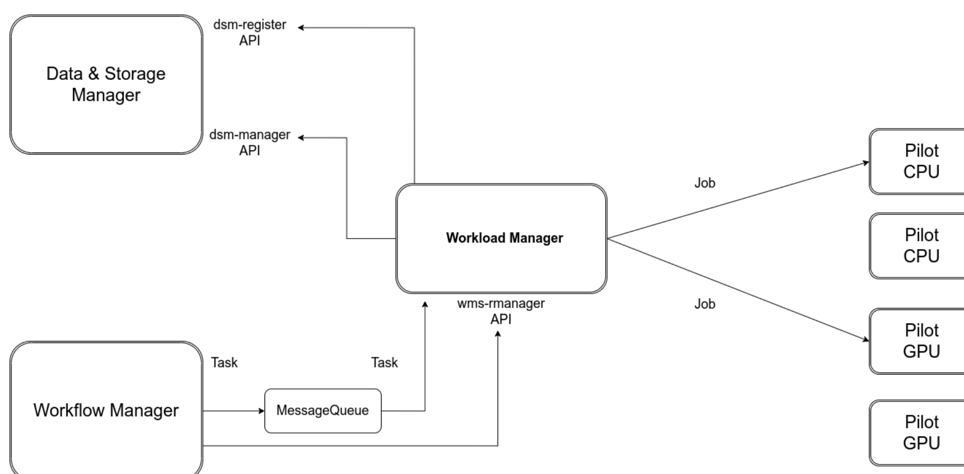

Рис. 5. Основные составляющие SPD Online Filter.

Для реализации всей системы предполагается разработка следующих компонент (рис. 5):

- Система управления данными (регистрация новых данных, каталогизация/структуризация, контроль целостности)
- Система управления процессом (составление набора заданий для каждого шага процесса обработки)
- Система управления нагрузкой, которую условно можно разделить на северную часть, отвечающую за контроль обработки наборов данных путем формирования достаточно количества задач и агентское приложение, пилот, которое обеспечивает выполнение задач на вычислительных узлах.

Постановка задачи

Основным моим направлением на проекте является разработка агентского приложения. Целью данной дипломной работы послужило проектирование данной системы, куда вошли следующие задачи:

- Изучение организации многоступенчатой обработки потока данных в системе «SPD On-Line filter»;
- Изучение существующих решений для агентских приложений;
- Проектирование внутренних и внешних интерфейсов взаимодействия пилотного приложения и серверной компоненты системы управления нагрузкой ;
- Спроектировать и оптимизировать архитектуру пилота;
- Выбор стека технологий и прототипирование;

Глава 1. Анализ существующих решений

Система управления нагрузкой (Workload Management System) – это важный компонент для управления высокопропускной обработкой данных в распределенной вычислительной среде. WMS автоматизирует процессы массовой обработки и моделирования данных. Кроме того, некоторые системы поддерживают настраиваемые процессы обработки для индивидуальных пользователей и обеспечивают единое представление глобально распределенных ресурсов.

Чтобы пользователи могли легко отслеживать текущий статус и историю каждого процесса обработки, WMS часто интегрирована с системой мониторинга. Система хранит и сопровождает все процессы обработки, управляет распределением данных, а также обеспечивает надежную и эффективную систему управления нагрузкой.

Пилоты являются составной частью системы управления нагрузкой и отвечают за выполнение задач на вычислительных узлах. Они обеспечивают захват ресурса и коммуникацию для получения задач, организацию их выполнения и передачу различной информации о ходе выполнения и состоянии вычислительного узла. Пилоты обеспечивают достаточную гибкость при управлении ресурсами двумя способами:

- с помощью поздней привязки, чтобы упростить и повысить эффективность управления приоритетами и как следствие возможность оптимизации пропускной способности [5-7];
- путем отделения спецификации рабочей нагрузки от управления ее исполнением.

Поздняя привязка приводит к возможности динамического использования ресурсов. Разделение спецификации рабочей нагрузки и ее выполнения упрощает планирование рабочих нагрузок на этих ресурсах.

Пилотные задания используются для потребления более 700 миллионов процессорных часов в год [8-9] сообществами Open Science Grid (OSG) [10-11] и обрабатывают до 1 миллиона заданий в день [12] для эксперимента ATLAS [13] на Большом Адронном Коллайдере (LHC) [14] посредством Worldwide LHC Computing Grid (WLCG) [15-16]. В инфраструктурах распределенных вычислений (DCI) используются различные системы пилотных заданий: Glidein/GlideinWMS [17-18], система Coaster [19], DIANE [20], DIRAC [21], PanDA [22], GWPilot [23], Nimrod/G [24], Falkon [25], MyCluster [26] и это лишь некоторые из них. Применение пилотной парадигмы в промежуточном ПО для распределенных вычислений стандартная практика, что демонстрирует рисунок 6.

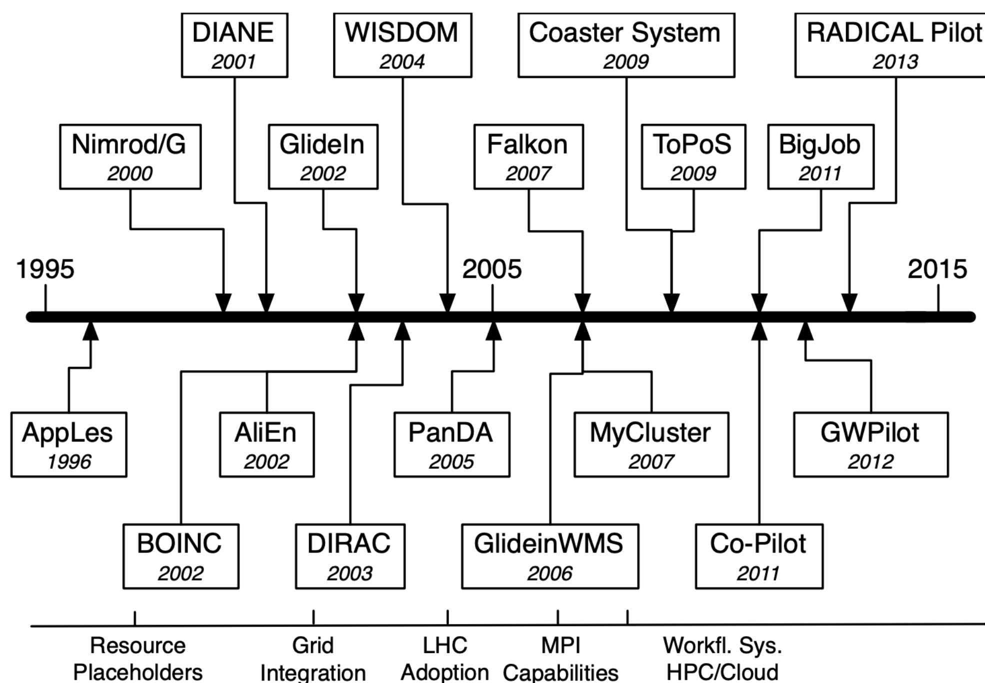

Рис. 6. Возникновение систем пилотных заданий с течением времени.

1.1 PanDA Pilot

Production and Distributed Analysis (PanDA) — это система управления рабочими нагрузками (WMS), способная работать в больших масштабах для

обработки данных и обеспечивающая гибкость адаптации к новым вычислительным технологиям в области обработки, хранения, сетевого взаимодействия и промежуточного программного обеспечения для распределенных вычислений. WMS-PanDA состоит из нескольких подсистем, PanDA Server, Harvester, PanDA Pilot, JEDI, PanDA Monitoring и ряда других (рис. 7). WMS-PanDA использует пилотные приложения для выполнения задач в среде грид, добровольных вычислительных системах, облачных инфраструктурах и суперкомпьютерах и поддерживает различные типы рабочих нагрузок.

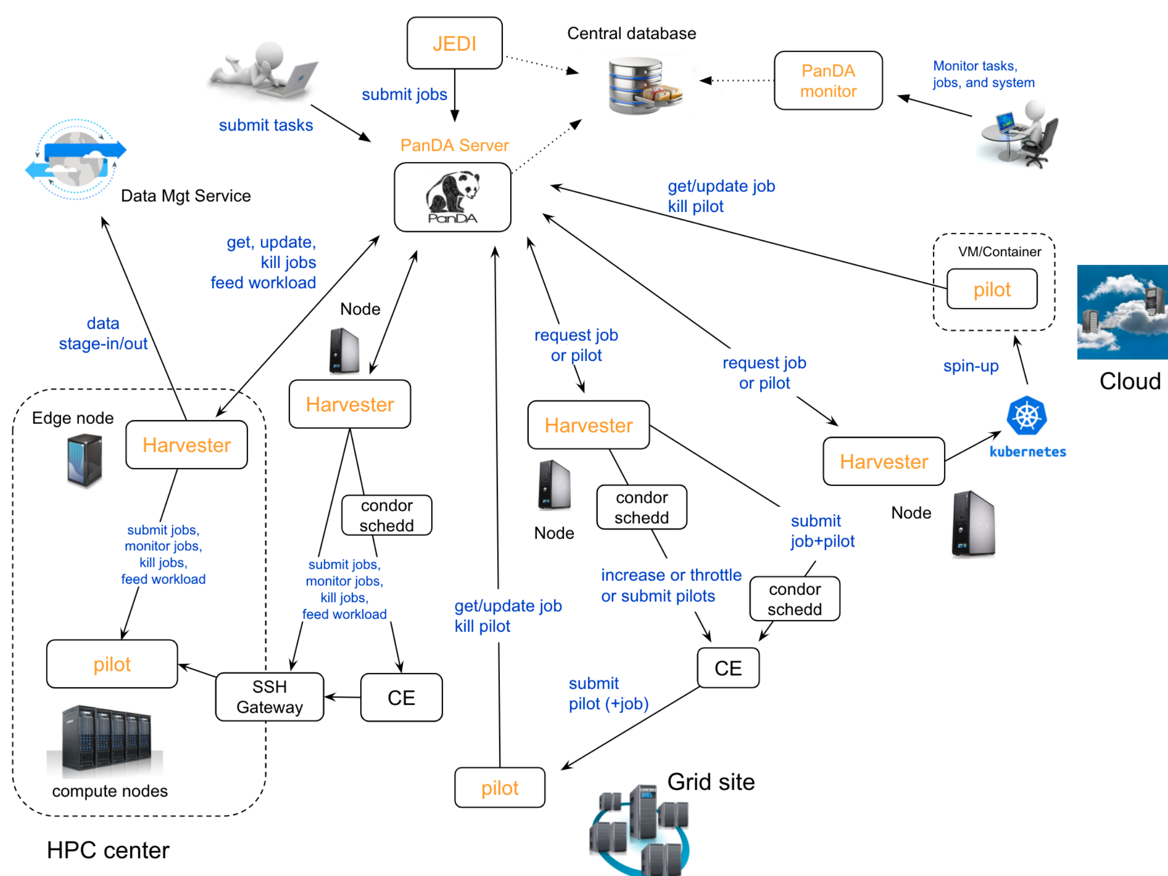

Рис. 7. Обзор системы PanDA.

Пилот должен выполнять и осуществлять мониторинг задач (Job) на вычислительных узлах. Задача – это атомарная единица работы (полезной нагрузки) возникающая как от систем массовой обработки так и от конечных

пользователей. К описаниям задач предъявляются определенные требования, например, декларация входных и выходных файлов, которые подготавливаются пилотом, или к рабочей среде, которую пилот должен настроить. Задача может потребовать запуска внутри контейнера, который также настраивается пилотом или, в некоторых случаях, скриптом, запускающим пилот. При детальной обработке пилот запускает и передает задачу с набором метаданных из вложенных файлов, загруженных с сервера PanDA.

Пилотное приложение запускается на рабочих узлах грид ресурсов, на суперкомпьютерах и на добровольных вычислительных системах. Он загружается и выполняется с помощью запускающего скрипта, которые отправляются специализированным сервисом на рабочие узлы через системы пакетной обработки. Пилот взаимодействует с сервером PanDA либо напрямую, через локальный экземпляр ARC Control Tower (платформа управления заданиями, используемая в Nordugrid), либо с ресурсоориентированной службой Harvester. Ниже перечислены некоторые основные моменты пилотного приложения:

1. Пилотное приложение отвечает за запуск полезных нагрузок, созданных пользователями или производственной системой, одновременно отслеживая все этапы и обновляя сервер PanDA.
    a. Все необходимые входные файлы и созданные выходные файлы будут перенесены в соответствующий элемент хранения.
    b. Полезная нагрузка может получить доступ к входным файлам непосредственно из хранилища, и в этом случае пилот не будет передавать файл.
    c. Полезная нагрузка может включать в себя процедуры предварительной подготовки, процедуры сопровождения выполнения и процедуры постобработки, а также самой основной полезной нагрузки.

   d. Пилот может выполнять специальные служебные процессы, например инструменты мониторинга памяти, работающие параллельно с полезной нагрузкой.
   e. Все процессы могут выполняться в своих собственных контейнерах, либо предопределенных, либо установленных пользователями.
2. Все коммуникации между pilot и внешними сервисами (например, PanDA Server, Rucio и CRIC) осуществляются с использованием защищенного протокола HTTPS.
3. Передача файлов осуществляется с помощью специальных средств копирования.
   a. В настоящее время поддерживаются такие инструменты копирования, как Rucio (с использованием Rucio API), xrdcp, gfal, gs, s3, mv/cp/ln, objectstore, lsm (локально определяемый движок сайта).
   b. Для каждой передачи файла (а также для файлов, к которым осуществляется прямой доступ) пилот может отправить подробный отчет на сервер трассировки Rucio.
4. HPC, не имеющие исходящей сети, поддерживаются путем делегирования связи прокси-службам.
5. Идентификация и сообщение о более чем 130 уникальных ошибках, включая подробную диагностику ошибок, когда это возможно. Можно настроить обработку ошибок и сообщений.
6. Существует множество вариантов отладки проблемных полезных нагрузок.
   a. Режим отладки может быть активирован при создании задания или задачи или после их запуска.
   b. За отдельным заданием, запущенным в режиме отладки, можно следить на PanDA Monitor по последнему измененному файлу,

загружаемому при каждом обновлении сервера (каждые пять минут в режиме отладки), или в режиме реального времени, регистрируя задачу практически в режиме реального времени с помощью Google Cloud Logging, Fluentd и Logstash.

   c. Пилот может выполнить ***ls*** (утилита Unix, которая печатает в стандартный вывод содержимое каталогов) в запрошенном рабочем каталоге и ***ps*** (утилита мониторинга работающих процессов в Linux) для заданного идентификатора процесса, а также отчитаться об использовании диска и сделать выходные данные доступными на PanDA Monitor. Он также может запустить отладчик gdb для создания основных файлов и размещения их в журнале заданий для последующего извлечения.

7. PanDA WMS в настоящее время используется для обеспечения обработки данных для целого ряда экспериментов: ATLAS, sPHENIX, LSST/Vera C. Rubin, COMPASS, SPD
   a. Пилотное приложение было адаптировано, для выполнения задач каждого из перечисленных экспериментов
   b. Специфика работы с задачами экспериментов вынесена в подключаемые модули.
8. Текущая пилотная версия совместима с Python 3.
   a. Запрос на загрузку в пилотный репозиторий GitHub запускает модульные тесты и запускает проверку Flake8 для версий Python 3.7, 3.8 и 3.9.

Пилот состоит из контроллеров, каждый из которых отвечает за разные задачи. Основные задачи выполняются контроллерами Job Control, Payload Control и Data Control. Существует также набор компонентов со вспомогательными функциями, например, Pilot Monitor и Job Monitor.

Опишем суть этих контроллеров:

Job Control – управляет Job, порождает пять подпотоков для различных задач (получение задачи из какого-либо источника, валидация входных параметров, мониторинг зависания задачи и т. д.);

Payload Control – основной элемент управления выполнения полезной нагрузки;

Data Control – элемент управление данными, которые необходимы для выполнения задачи и выгрузки результатов;

Pilot Monitor – для внутреннего использования, отслеживает потоки;

Job Monitor – привязан к заданию и проверяет параметры, относящиеся к payload (например, проверки размера).

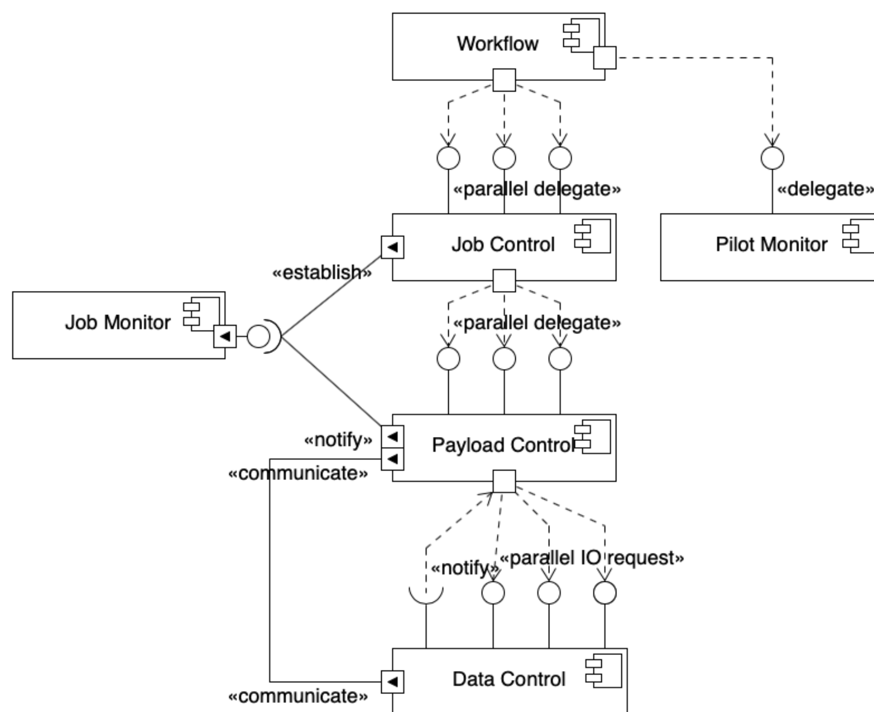

Рис. 8. Общая архитектура.

Каждый из функциональных компонентов работает как независимый поток, а взаимодействие между потоками осуществляется через стеки сообщений. Каждый компонент имеет дополнительные потоки:

1. Job Control

Название функционала	Описание
retrieve	Получить определение задания из источника и поместить его в очередь «job». Определение задания представляет собой словарь json, который предварительно помещается в каталог запуска или загружается с сервера, указанного в args.url.
validate	Получить объект Job из очереди «jobs». Если он проходит проверку, определяемую пользователем, создается основной рабочий каталог полезной нагрузки (PanDA_Pilot-<pandaid>) в основном рабочем каталоге пилотного проекта. Объект задания передается в очередь «validated_jobs» или «failed_jobs» в случае сбоя.
create_data_payload	Получить объект Job из очереди «validated_jobs». Если задание имеет определенные входные файлы, переместить объект задания в очередь «data_in» и установить внутреннее пилотное состояние в «stagein». Если входных файлов нет, поместите объект Job в очередь «finished_data_in». В любом случае поток также помещает объект задания в очередь «payloads».
queue_monitor	Мониторинг очередей. Этот поток отслеживает активность очереди, в частности, завершение или сбой задания, и сообщает серверу. Завершенная

	работа будет перемещена в очередь «completed_jobs».
job_monitor	Мониторинг параметров работы. Этот поток отслеживает определенные параметры задания, например зацикливание задания, через различные промежутки времени. Основной цикл выполняется раз в минуту, а отдельные проверки могут выполняться с любым интервалом времени (>= 1 минута).

2. Payload Control

Название функционала	Описание
validate_pre	Получите объект Job из очереди «payloads». Если полезная нагрузка успешно проверена (определена пользователем), объект Job помещается в очередь «validated_payloads», в противном случае он помещается в очередь «failed_payloads».
execute_payloads	Извлечь объект Job из очереди «validated_payloads» и поместить его в очередь «monitored_jobs». Открываются потоки полезной нагрузки stdout/err, а состояние пилот-сигнала изменяется на «starting». Выбирается исполнитель полезной нагрузки (для выполнения обычного задания, задания службы событий или задания

	объединения службы событий). После запуска полезной нагрузки (точнее, ее исполнителя) поток будет ждать ее завершения, а затем проверять наличие сбоев. Успешно выполненное задание помещается в очередь «finished_payloads», а невыполненное задание помещается в очередь «failed_payloads».
validate_post	Проверка готовых полезных нагрузок. Завершенное задание будет добавлено в очередь "data_out".
failed_post	Получить объект Job из очереди «failed_payloads». Установить для пилотного состояния значение «stakeout», а для поля stageout — «log» и добавить объект «Job» в очередь «data_out».

3. Data Control

Название функционала	Описание
copytool_in	Вызвать функцию stage-in и поместить объект Job в соответствующую очередь. Получить объект Job из очереди «data_in» и сразу же поместить в «current_data_in». Сообщить серверу, что задание находится в состоянии «running» (обратите внимание, что полезная нагрузка еще не запущена, а задание запущено). Вызывается stage-in функция, и, если она завершается правильно,

	объект Job перемещается в очередь «finished_data_in», а также удаляется из очереди «current_data_in». В случае сбоя stage-in функуии объект задания перемещается в очередь «failed_data_in».
copytool_out	Выполнить stage-out вывод, как только объект задания можно будет извлечь из очереди data_out. Если функция stage-out завершения завершается правильно, поместите объект задания в очередь «finished_data_out». В случае неудачи поместить в очередь «failed_data_out»
queue_monitoring	Мониторинг очередей данных. Если объект Job можно извлечь из очереди «failed_data_in» (или из очереди «failed_data_out»), установить для поля stage-out значение «log» и попытаться вывести журнал. Если объект задания можно извлечь из очереди «finished_data_out», поместить успешное задание в очередь «finished_jobs», а неудачное задание — в очередь «failed_jobs».

Большинство потоков манипулируют объектом Job, которые содержат всю полную информацию о задаче, загруженном с сервера PanDA или прочитанном из файла. Неполный пример объекта Job приведен на рисунке 9. Общее количество параметров в классе JobData составляет около 130.

```python
class JobData(BaseData):
    """
    High-level object to host Job definition/settings
    """

    # ## put explicit list of all the attributes with comments for better inline-documentation by Sphinx
    # ## FIX ME LATER: use proper doc format
    # ## incomplete list of attributes .. to be extended once becomes used

    jobid = None                    # unique Job identifier (forced to be a string)
    parent_jobid = None
    script_name = ""
    taskid = None                   # unique Task identifier, the task that this job belongs to (forced to be a string)
    batchid = None                  # batch system job id (should be removed from here)
    batchtype = None                # batch system type (should be removed from here)
    jobparams = ""                  # job parameters defining the execution of the job
    transformation = ""             # script execution name
    # current job status; format = {key: value, ..} e.g. key='LOG_TRANSFER', value='DONE'
    status = {'LOG_TRANSFER': LOG_TRANSFER_NOT_DONE}
    corecount = 1                   # Number of cores as requested by the task
    platform = ""                   # cmtconfig value from the task definition
    transfertype = ""               # direct access instruction from server
    accessmode = ""                 # direct access instruction from jobparams
    processingtype = ""             # e.g. nightlies
    maxcpucount = 0                 # defines what is a looping job (seconds)
    allownooutput = ""              # used to disregard empty files from job report
    realtimelogging = False         # True for real-time logging (set by server/job definition/args)
    pandasecrets = ""               # User defined secrets
    pilotsecrets = {}               # Real-time logging secrets
```

Рис. 9. Пример некоторых данных объекта Job.

Экземпляр объекта Job хранится в глобально доступных очередях Python и передается по разным очередям до тех пор, пока его обработка не завершится. Различные потоки наблюдают за этими очередями и воздействуют на объект Job по мере его поступления.

Каждый поток опрашивает очередь до тех пор, пока не получит объект Job для обработки; после обработки результат помещается в другую очередь для дальнейшей обработки, и поток снова начинает опрашивать свою входную очередь. Сам объект задания представляет собой сущность, которая содержит всю необходимую информацию о запуске полезной нагрузки, такую как версия программного обеспечения, параметры для настройки payload, тип передачи входных файлов и т. д.

В частности, в этом рабочем процессе Pilot выполняет загрузку payload (полезной нагрузки), настройку, ввод, выполнение и вывод, а также различные проверки, мониторинг и обновление заданий сервера через выбранные интервалы времени.

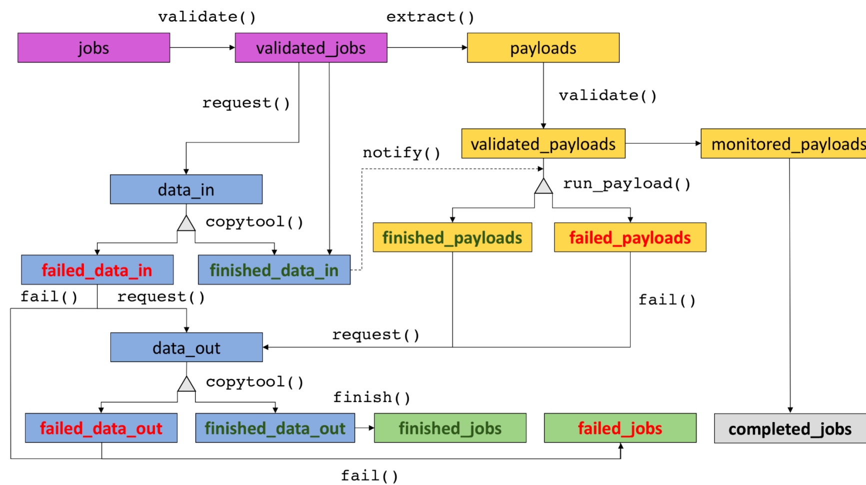

Рис. 10. Стандартный рабочий процесс.

На рисунке выше показан внутренний общий поток объектов задания. Загруженное задание вставляется в очередь заданий и попадает либо в очередь finish_jobs, либо в очередь failed_jobs. Очередь complete_jobs содержит копию объекта задания, выполнение которого завершилось (завершено или завершилось с ошибкой), и используется для внутреннего учета.

В обзоре решения PanDA Pilot [27] была рассмотрена архитектура, которая имеет свои преимущества и недостатки. Кодовая база проекта составляет порядка 25.000 строк кода, который реализует много действий и учитывает большое количество разных сценариев: HPC Workflow, Event service, Direct Access и т. д. Таким образом, несмотря на то, что базовая архитектура пилота хорошо продумана, в качестве готового или частичного решения для SPD она не подходит: кодовая база слишком велика и помимо этого она сильно зависит от конкретной архитектуры PanDA WMS.

1.2 DIRAC Pilot

DIRAC (Distributed Infrastructure with Remote Agent Control) [28-29] – представляет собой систему, обеспечивающую основу для распределенных вычислений и управления данными для различных научных сообществ

(физика элементарных частиц, астрофизика, космология и т.д.). Состоит из нескольких сервисов, которые взаимодействуют друг с другом и с внешними ресурсами. Одним из ее компонентов является система управления рабочей нагрузкой (WMS), которая обрабатывает отправку и выполнение задач (jobs) на различных типах ресурсов, включая кластеры высокопроизводительных вычислений. Также содержит таким компоненты как система управления данными (DMS), система мониторинга, система конфигурации и т. д.

Каждое пилотное задание DIRAC (DIRAC Pilot Job) выполняет установку на месте, включая полную загрузку самой последней версии конфигурации. В этой новой установке и после проверки условий работы (точное местоположение, где выполняется выполнение, доступное дисковое пространство, память, центральный процессор, доступная сетевая среда, запущенная платформа) выполняется агент задания (Job Agent) DIRAC. Этот агент задания отвечает за отправку запроса полезной нагрузки на центральный сервер DIRAC WMS и за последующее выполнение полученной полезной нагрузки.

Чтобы обеспечить общую среду выполнения для всех полезных нагрузок, Job Agent создает экземпляр Job Wrapper. В то же время он также создает экземпляр Watchdog для контроля за правильным поведением Job Wrapper. Watchdog периодически проверяет состояние оболочки, предпринимает действия в случае, если диск или доступный процессор вот-вот будут исчерпаны или полезная нагрузка остановится, и сообщает об этом в центральную WMS. Он также может выполнять команды управления, полученные от центральной WMS, например, прервать выполнение задачи. Job Wrapper извлекает входную изолированную среду, проверяет доступность необходимых входных данных и программного обеспечения, выполняет полезную нагрузку, сообщает об успешном или неудачном выполнении и, наконец, загружает выходную изолированную среду и выходные данные, если требуется.

Этот механизм позволяет пользователям сконцентрировать свои усилия на определении реальной полезной нагрузки, поскольку DIRAC берет на себя бремя всех этих дополнительных шагов, которые определяют разницу между локальным и удаленным выполнением. Оболочка DIRAC единообразно заботится обо всех этих деталях, предоставляя пользователю четко определенную и общую среду выполнения на разных вычислительных ресурсах.

Чтобы эффективно определить наиболее подходящую полезную нагрузку, которая будет выполнена пилотным заданием в любой момент времени, отложенные задачи организованы в очереди задач (TaskQueues).

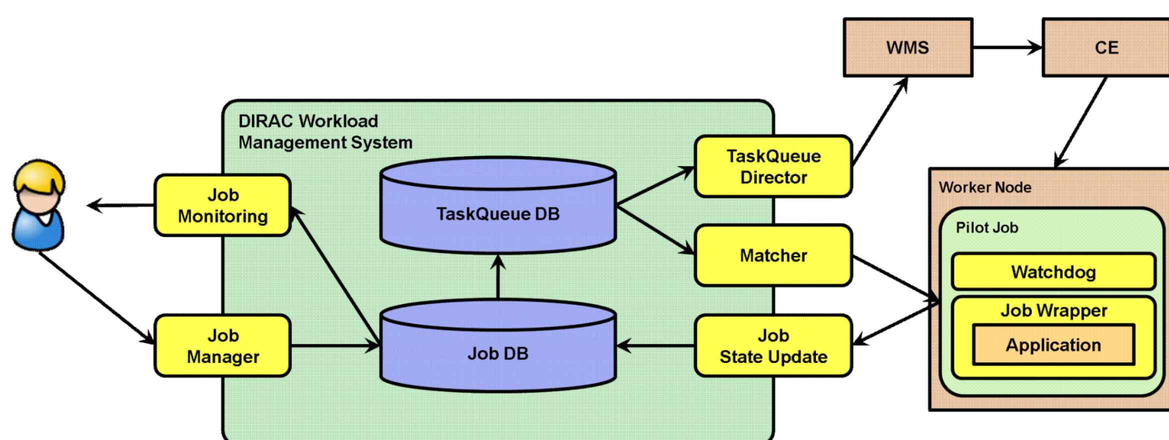

Рис. 11. Общая архитектура DIRAC Pilot.

Весь процесс может быть описан следующей схемой, представленной на рисунке 11. С левой стороны пользователь инициирует всю процедуру, отправляя новую полезную нагрузку. Для простоты показан только один директор (Director), отправляющий данные в WLCG через серверы gLite WMS. После отправки описание полезной нагрузки вставляется в хранилище заданий DIRAC, JobDB. Это описание используется для добавления записи в TaskQueueDB, добавляя полезную нагрузку к существующей подходящей очереди задач. Если подходящая очередь не найдена, то создается новая. Основываясь на полном списке ожидающих выполнения полезных нагрузок,

Director Agents используются для заполнения доступных вычислительных ресурсов пилотными заданиями. После развертывания системы DIRAC и проверки локальной среды на WN (Worker Node) пилоты запрашивают полезную нагрузку на центральный сервер DIRAC WMS. Наконец, полезная нагрузка выполняется внутри оболочки DIRAC под наблюдением Watchdog DIRAC.

Пилот тесно интегрирован в программную платформу Dirac, в результате чего увеличивается размер приложения и может достигать 1,5 Гб.

1.3 RADICAL-Pilot

RADICAL-Pilot создан на основе опыта разработки BigJob и его интеграции со многими приложениями [30-31] на различных DCR (Distributed Computing Resource).

RADICAL-Pilot состоит из пяти основных логических компонентов (рис. 12): Pilot Manager, Compute Unit (CU) Manager, набора Agents, интерфейса SAGA-Python DCR и базы данных. Pilot Manager описывает пилотные задания и отправляет их через SAGA-Python в DCR, в то время как менеджер CU описывает задачи (т. е. CU) и назначает их одному или нескольким пилотам. Агенты создаются на DCR и выполняют CU, выдвинутые менеджером CU. База данных используется для связи и координации остальных четырех компонентов.

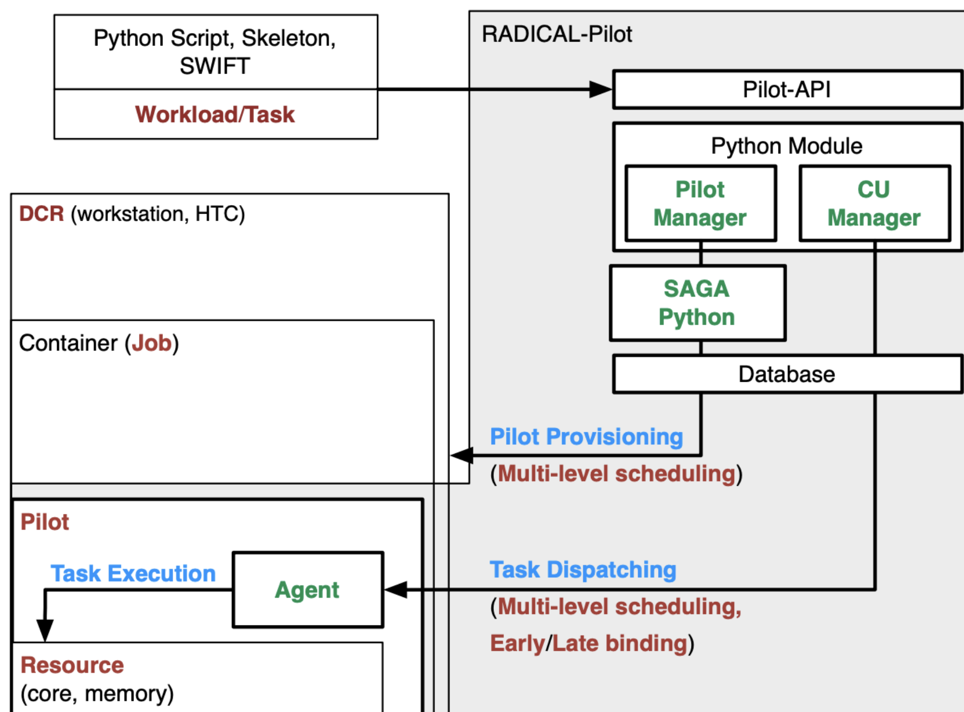

Рис. 12. Схематическое представление компонентов, функциональных возможностей и основной терминологии RADICAL-Pilot.

Pilot Manager и SAGA-Python реализуют логический компонент. Диспетчер рабочей нагрузки реализуется менеджером CU. Агент развертывается в DCR для предоставления своих ресурсов и выполнения задач, поставленных менеджером CU. Таким образом, Агент является Пилотом.

RADICAL-pilot реализован в виде двух модулей Python для поддержки разработки распределенных приложений. Выполнение RADICAL-Pilot можно обобщить в шесть шагов:

1. Пользователь описывает задачи в Python как набор CU с данными (или без них) и зависимостями DCR;

2. Пользователь также описывает одного или нескольких пилотов, которым должны быть отправлены CU;

3. При выполнении заявки пользователя Pilot Manager отправляет каждый описанный пилот в указанный DCR, используя интерфейс SAGA;

4. CU Manager назначает каждый CU либо на пилот, указанный в CU, либо на первый пилот со свободными и доступными ресурсами. Планирование выполняется путем сохранения описания CU в базе данных;

5. При необходимости менеджер CU также передает входной файл(ы) CU в целевой DCR;

6. Агент извлекает свой CU из базы данных и выполняет его.

Агентный компонент RADICAL-Pilot предлагает абстракции как для вычислительных ресурсов, так и для ресурсов данных. Каждый агент может предоставлять от одного до всех ядер вычислительного узла, на котором он выполняется; он также может предоставлять дескриптор данных, который абстрагирует определенные свойства и возможности хранилища. Таким образом, CU, работающие на агенте, могут использовать унифицированные интерфейсы как для ядра, так и для ресурсов данных. Предполагается, что сетевое взаимодействие между компонентами RADICAL-Pilot доступно.

Pilot Manager развертывает агенты RADICAL-Pilot с помощью API SAGA-python [32]. SAGA предоставляет доступ к разнообразному промежуточному программному обеспечению DCR через унифицированный и согласованный API.

Полученное в результате отделение развертывания агента от архитектуры DCR снижает накладные расходы на добавление поддержки нового DCR [33]. Это иллюстрируется относительной легкостью, с которой RADICAL-Pilot расширяется для поддержки:

- нового типа DCR, такого как IaaS,
- DCR, которые имеют по существу аналогичную архитектуру, но другое промежуточное программное обеспечение, например, суперкомпьютеры Cray, работающие в США и Европе.

RADICAL-Pilot может выполнять задачи с различными требованиями к соединению и связи. Задачи могут быть полностью независимыми, однопоточными или многопоточными; они могут быть слабо связаны, требуя

зависимостей между входными и выходными файлами, или могут требовать взаимодействия с малой задержкой во время выполнения. Таким образом, RADICAL-Pilot поддерживает приложения MPI, рабочие процессы и различные шаблоны выполнения, такие как конвейеры.

Описания CU могут содержать или не содержать ссылку на пилотную программу, к которой пользователь хочет привязать CU. Когда ссылка присутствует, планировщик CU Manager ожидает, пока слот не будет доступен для указанного пилота. Если целевой пилот не указан, диспетчер CU связывает и планирует CU на первом доступном пилоте. Таким образом, RADICAL-Pilot поддерживает как раннее, так и позднее связывание, в зависимости от варианта использования и спецификаций пользователя.

Пилот является базовым компонентом в системах управления нагрузкой. Его реализация всегда специфична и тесно связана с архитектурой и реализацией всей системы. Поэтому, невозможно взять пилот из одной системы и путем минимальных преобразований заставить работать в другой.

Глава 2. Общая архитектура

2.1 Workload Management System (WMS)

Перед тем, как говорить про сам пилот, нужно пояснить основные моменты архитектуры **Workload Management System (WMS)** (рис. 13) – единственной системой, с которой взаимодействует пилотное приложение.

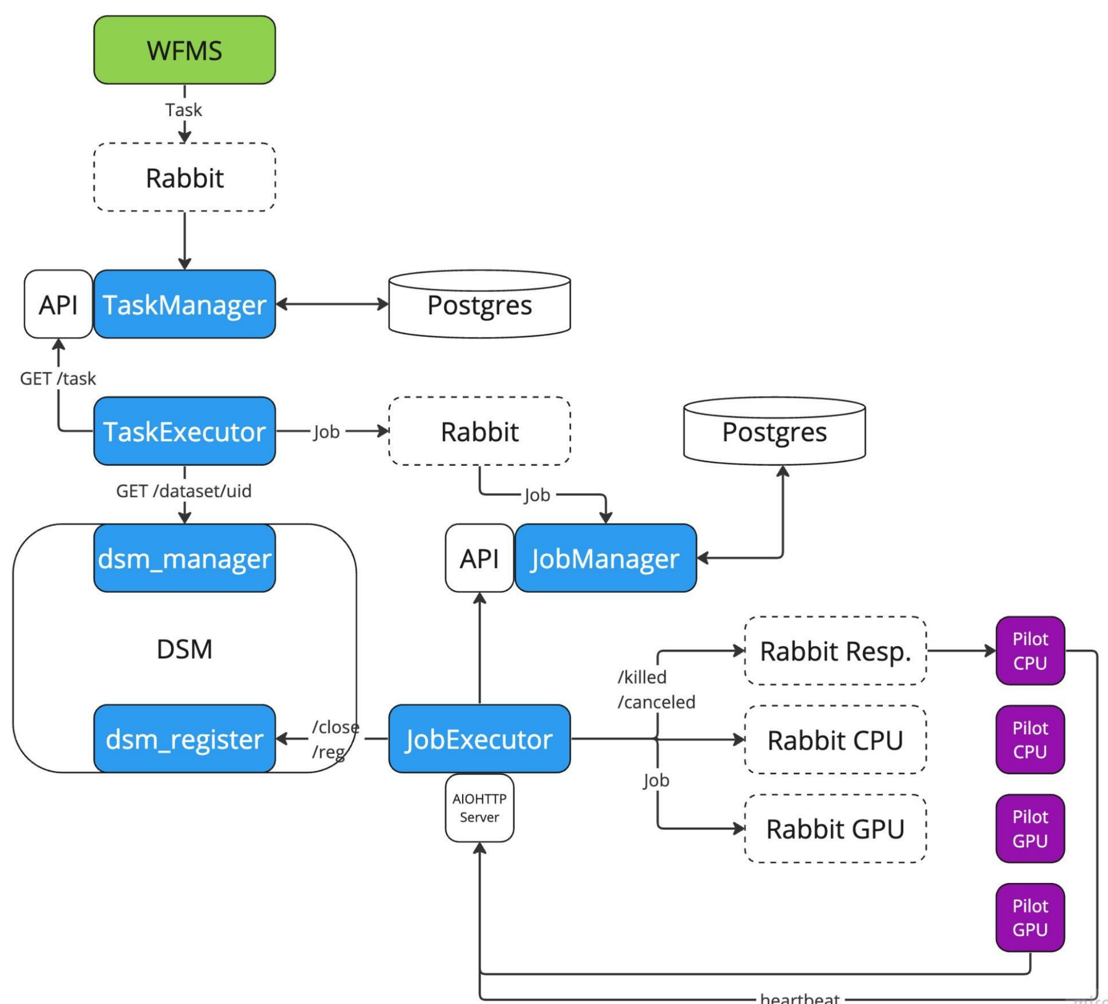

Рис. 13. Архитектура системы управления нагрузкой.

Жизненный цикл системы выглядит следующим образом:

- На стороне **Workflow Management System** идёт проверка готовности датасета в **DSM** для последующей обработки. В случае успеха создается шаблон задания, который в дальнейшем передается в очередь RabbitMQ.

- **Task-Manager** принимает из очереди заданий (*direct exchange*) в формате JSON, добавляет в таблицу заданий, и задает статус готовности для приема на стороне **Task-Executor**.
- **Task-Executor** запрашивает следующее готовое задание для обработки. После его получения, микросервис запрашивает у **dsm-manager** структуру датасета. Затем начинается процесс генерации задач из расчета “одна задача на один файл”, что происходит в асинхронном порядке по нескольким датасетам. Другими словами происходит “дробление” датасета на более мелкие единицы рабочего процесса (*Map*-шаг).
- Сформированные задачи отправляются в очередь сообщений RabbitMQ на **Job-Manager**.
- **Job-Manager** получает описания задач из очереди, добавляет по полученным описаниями соответствующие кортеже в таблицы задач, файлов и заданий. Следит за согласованностью и текущим статусом задач. Формирует имена выходных файлов и логов.
- **Job-Executor** запрашивает следующую задачу, определяет свободные/подходящие пилоты, ресурсы и посылают в соответствующие очереди на пилоты (распределяет задачи на пилоты согласно его текущей политике). В случае простоя пилотов, определяет следующую по приоритету задачу, которая пойдет в очередь.
- Пилот присылает регулярные сообщения о статусе задачи, **Job-Executor** обновляет статус задачи, регистрирует завершившиеся задачи, закрывает датасет в случае обработке всех задач либо посредством запроса от **WFMS**.

2.2 Формирование задач

Задача в определенном смысле – единица рабочей нагрузки, порожденная от задания. Выполнение одного задания заключается в выполнении достаточного набора однородных задач, и каждая задача выполняется на минимально необходимом наборе вычислительных ресурсов (одна задача на один пилот). Входные датасеты каждого задания разбиваются на несколько непересекающихся подмножеств, и каждая задача получает данное подмножество для дальнейшей работы. Совокупность выходных данных в результате работы всех задач — это выходные данные всего задания.

2.3 Жизненный цикл задач (Jobs)

Каждая задача начинает свое существование в системе в **TaskExecutor/Shredder** после получение метаданных о файле от сервиса ***dsm-manager***, и заканчивает свое существование после регистрации соответствующего ей выходного файла с результатом и логом. Ниже схематично показан граф смены состояния задачи (рис. 14).

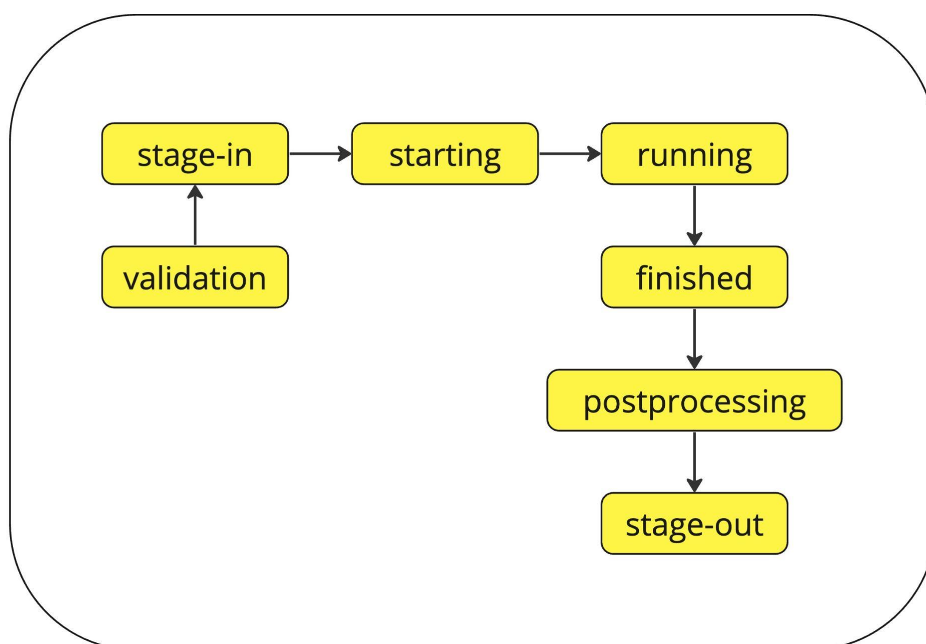

Рис. 14. Жизненный цикл задачи на пилоте.

2.4 Взаимодействие WMS и Pilot

Аллокация перестаёт иметь первостепенное значение, потому что весь ресурс и пилоты «прибиты» к вычислительным узлам и пилотам остаётся сосредоточиться только на выполнении своих задач. За назначение задач отвечает планировщик задач, который занимается долгосрочным планированием (long-term scheduling). Сам пилот взаимодействует только с JobExecutor. Ниже приведены возможные варианты взаимодействия между JobExecutor и пилотами:

1. Пилот запускается, берет задачу из очереди. Если задача есть, то посылает сообщение со статусом.
2. Пилот отправляет heartbeat-сообщения на **Job-Executor** с заданным периодом. Передает: идентификатор пилота, параметры вычислительного узла (CPU, CPU+GPU). Возвращает статус задачи.
3. Пилот регулярно обновляет статус задачи по мере выполнения. Для этого создается ещё одна сессия aiohttp.
4. По окончанию работы передает: полный путь к выходным файлам (результат работы задачи, журнал задачи), код выхода задачи, сообщение ошибки (если есть), тип ошибки.

2.5 Job-executor

Job-executor – сервис, отвечающий за отправку задач в очереди на пилоты, регистрацию и закрытие датасета, также ответственен за обновление статуса задач. Описание задачи приведено в приложении листинг 1.

Глава 3. Архитектура VISOR Pilot

Учитывая специфику системы управления нагрузкой и условий реализации SPD Online Filter, было необходимо реализовать собственное пилотное приложение, которое бы удовлетворяло следующим ограничениям:

- Относительно гомогенная вычислительная среда.
- Вычислительные узлы различаются только наличием специализированных сопроцессоров (GPU). В остальном считаем, что они одинаковые.
- Необходимо иметь возможность смены версий пилотного приложения без остановки всей системы.
- Пилотное приложение должно реализовывать необходимый набор интерфейсов с серверной частью WMS.
- Жизненный цикл задачи выполняемой в пилотном приложении когерентен с жизненным циклом задачи в системе.

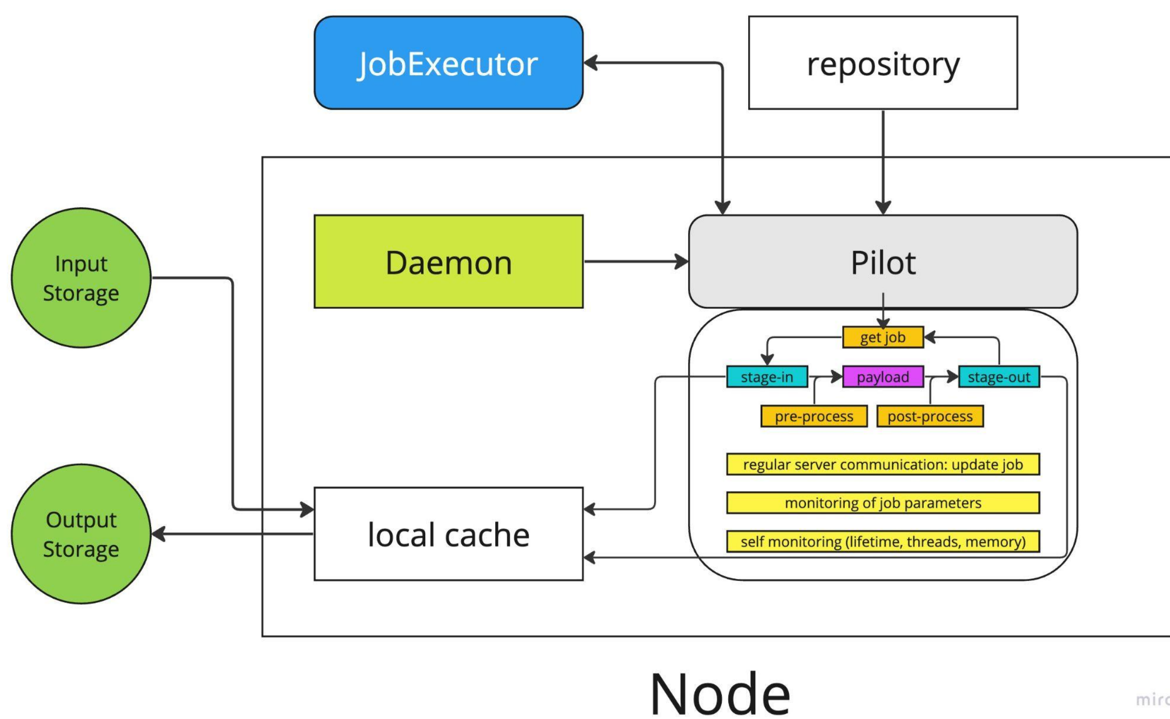

Рис. 15. Общая схема программного комплекса.

На рисунке 15 представлена общая схема программного комплекса, который состоит из 2-х компонент: демона и самого пилота. Демон запущен на узле постоянно и его задача заключается в том, чтобы запускать очередной пилот, скачивая актуальную версию из репозитория. Таким образом решается проблема смены версий пилотного приложения без остановки всей системы.

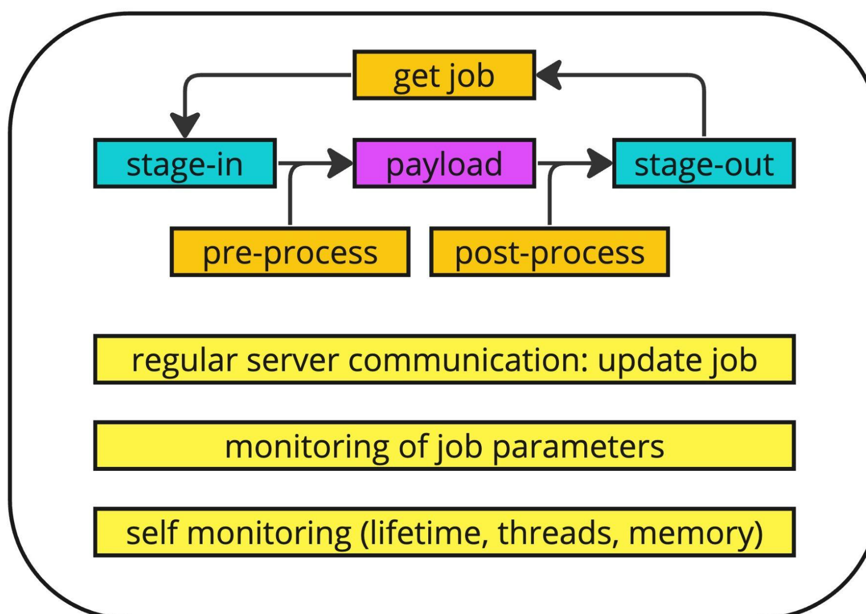

Рис. 16. Схема пилота.

На рисунке 16 представлена схема пилота. После запуска пилот берет из очереди новую задачу и выполняет ее. После успешного или неуспешного выполнения пилот завершает свою работу, а демон в свою очередь должен запустить новый экземпляр пилота. Таким образом решается задача замены версии пилота.

На схеме разными цветами выделены 4 главных контроллера: Job Control (фиолетовый), Data Control (синий), Payload Control (желтый), Monitor (зелёный). Каждый из них выполняется в отдельном потоке.

Важным моментом является тот факт, что вычислительные узлы бывают двух типов: CPU и CPU+GPU. Имеющиеся задачи также разделяются

в зависимости от вычислительных потребностей. Диспетчер отправляет задачи в две очереди (рис. 17), а пилот опрашивает эти очереди и берет оттуда задачу. При этом, если очередь с задачами GPU пуста, пилот должен взять задачу из очереди с CPU задачами.

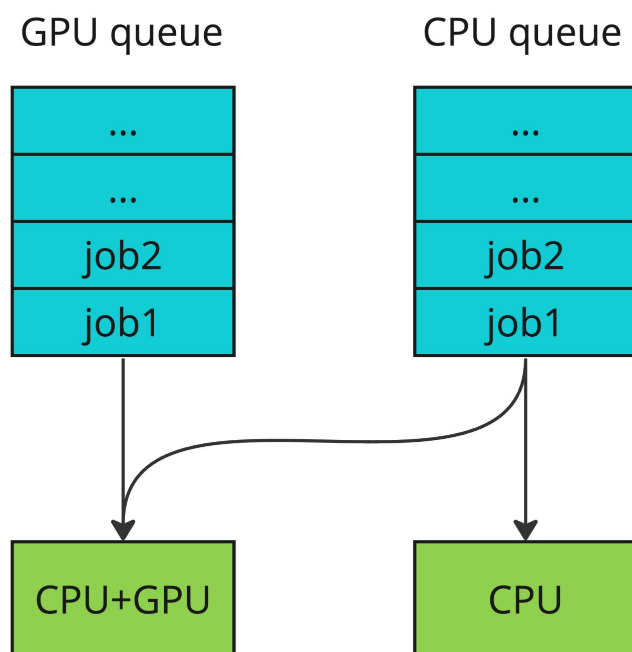

Рис. 17. Две очереди с задачами CPU и CPU+GPU.

Глава 4. Разработка агентского приложения

Разработка сервисов ведется в облачной инфраструктуе МЛИТ ОИЯИ. Репозитории с исходном кодом расположены на внутреннем GitLab организации. Их организация представлена на рисунке 18.

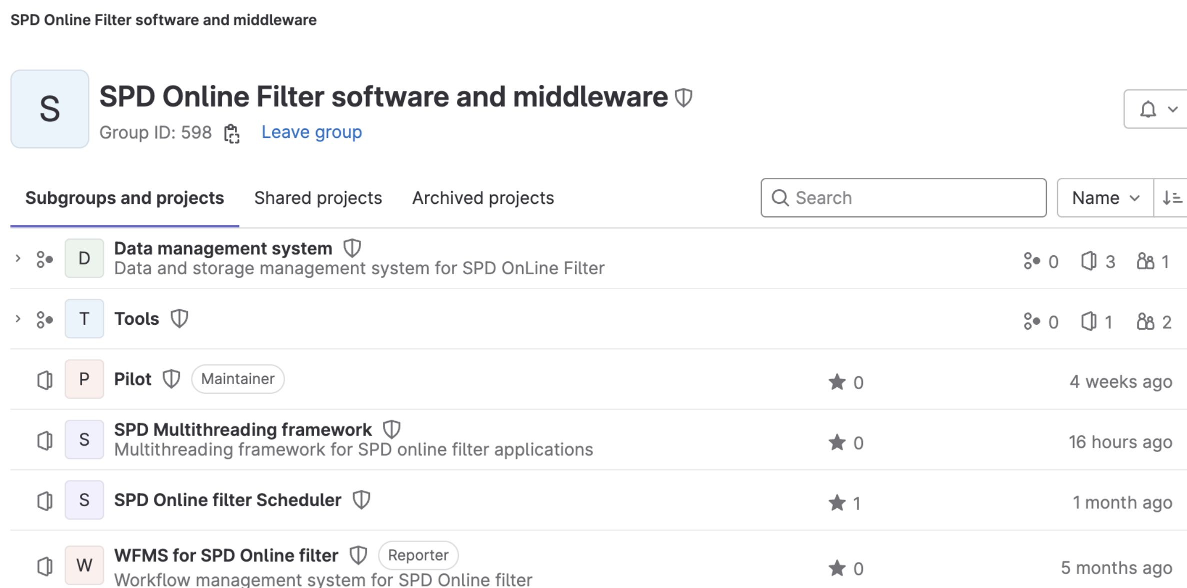

Рис. 18. Организация репозиториев с исходном кодом.

4.1 Выбор стека технологий

Для реализации использовался следующий стек технологий:

- Python 3 — основной язык программирования проекта;
- Conda — это кроссплатформенный менеджер пакетов с открытым исходным кодом и система управления средой, не зависящая от языка.
- Threading — библиотека, позволяющая работать с несколькими потоками;
- Aiohttp — асинхронный HTTP-клиент/сервер для модуля asyncio. Это библиотека языка Python, которая нужна для выполнения клиентских запросов и создания веб-сервера с потоковой выдачей и веб-сокетами.
- RabbitMQ — программный брокер сообщений на основе стандарта;

- AMQP — тиражируемое связующее программное обеспечение, ориентированное на обработку сообщений;
- Pika — это реализация протокола AMQP 0-9-1 на чистом Python, которая пытается оставаться достаточно независимой от базовой библиотеки поддержки сети.

4.2 Интерфейс взаимодействия с JobExecutor

Взаимодействие пилота и JobExecutor делятся на два типа и происходит посредством очередей брокера сообщений и протокола HTTP:

- Получение задач и управляющих команд осуществляется через брокер.
- Отправка статуса завершения и heartbeat-сообщений через протокол HTTP.

Метод	**API**	**Input**	**Output**
Взаимодействие через aiohttp			
Отправить heartbeat-сообщения	POST /job/<id>	ID + status + log_size	response
Окончание завершения задачи	POST /job/<id>/success	{files_path, status, job_log, error_code, error_type}	-
Взаимодействие через очереди сообщений от JobExecutor к пилотам			
Получить задачу CPU	JSON	Описан в приложении листинг 1	-
Получить задачу GPU	JSON	Описан в приложении листинг 1	-
Обновление статуса задачи	JSON	{killed/canceled}	-

4.3 RabbitMQ

RabbitMQ – это распределенный брокер сообщений с открытым исходным кодом, который используется разработчиками для обеспечения эффективной доставки сообщений в сложных сценариях маршрутизации. RabbitMQ часто используется в качестве кластера узлов, где очереди распределяются по узлам и реплицируются для обеспечения высокой доступности и отказоустойчивости. Благодаря этому, RabbitMQ может обеспечить надежную и стабильную работу в условиях высокой нагрузки.

RabbitMQ может использоваться для обработки высокопроизводительных фоновых задач, таких как батчи, интенсивные задачи обработки и длительные процессы workflow, а также для интеграции и взаимодействия между приложениями и внутри них.

Кроме этого, RabbitMQ имеет ряд дополнительных функций, которые могут быть полезными для разработчиков. Например, он поддерживает множество протоколов, включая AMQP, MQTT и STOMP, что позволяет использовать его в различных сценариях взаимодействия между системами. RabbitMQ имеет гибкую систему маршрутизации сообщений, которая позволяет настраивать его под конкретные потребности проекта.

Таким образом, RabbitMQ является мощным и гибким инструментом, который может быть использован для обеспечения эффективной доставки сообщений в условиях высокой нагрузки, а также для интеграции и взаимодействия между приложениями и внутри них.

4.4 Принимаемые значения статуса задачи

Статус задач (Jobs) фигурирует непосредственно только после момента их появления в системе **JobManager** и описывает текущее состояние выполнения задачи на пилоте, не путать с жизненным циклом задач.

Статус	*Описание*
Running	Задача обрабатывает входные данные.
Finished	Задача была успешно завершена.
Killed	Задание была намерено прервано.
Failed	Задача была неуспешно завершена на пилоте во время работы исполняемого файла.
Starting	Задача работает над перемещением входных данных - передача данных из хранилища на рабочий диск, подключенный к вычислительному ресурсу.
Cancelled	Задача была отменена “вручную”

Принимаемые значения статуса файла

Статус	*Описание*
Failed	Отсутствует выходной файл (по разным причинам)
Finished	Был получен ненулевой результат

4.5 Структура проекта

Структура проекта VISOR Pilot разработана с учётом удобства дальнейшего его сопровождения. Текущие её описание представлено на рисунке 19.

Рис. 19. Структура проекта.

В приложении листинг 2 приведен код запуска потоков. В нем происходит инициализация очередей, через которые проходит задача во время ее обработки.

Выводы

Проведен анализ существующих систем, определена общая архитектура приложения, формализованы и согласованы интерфейсы. Определены достаточный уровень функциональных требований и основные компоненты пилотной системы, способы взаимодействия системы с Workload Manager.

Заключение

В результате получен шаблон инструмента, позволяющий запускать простые задачи на кластере, а также отслеживать некоторые состояния. Разрабатываемый пилотный проект является критичным компонентом системы и требует дальнейшей доработки.

В дальнейшем планируется усовершенствование пилотного приложения, включая разработку новых модулей и оптимизацию существующих, доработать взаимодействие с остальными системами, провести полноценное тестирование на эмулированных данных и в конечном счёте начать интегрировать систему с прикладным ПО.

Список источников

[1] Эксперимент SPD [Электронный ресурс] // URL: http://spd.jinr.ru/wp-content/uploads/2021/04/SPD_Korzenev_DIS2021.pdf (дата обращения: 04.11.2022)

[2] NICA [Электронный ресурс] // URL: https://nica.jinr.ru/ru/ (дата обращения: 06.11.2022)

[3] Seven-Year Plan for the Development of JINR for 2024–2030 [Электронный ресурс] // URL: https://indico-hlit.jinr.ru/event/329/contributions/1995/attachments/578/1035/01_Director_next_7YP_for_LIT.pdf (дата обращения: 13.04.2023)

[4] Conceptual design of the Spin Physics Detector [Электронный ресурс] // URL: https://arxiv.org/abs/2102.00442 (дата обращения: 13.04.2023)

[5] J. Mo´scicki, M. Lamanna, M. Bubak, and P. M. Sloot, “Processing moldable tasks on the grid: Late job binding with lightweight user-level overlay,” Future Generation Computer Systems, vol. 27, no. 6, pp. 725–736, 2011.

[6] T. Glatard and S. Camarasu-Pop, “Modelling pilot-job applications on production grids,” in Proceedings of Euro-Par 2009 – Parallel Processing Workshops. Springer, 2010, pp. 140–149.

[7] A. Delgado Peris, J. M. Hernandez, and E. Huedo, “Distributed scheduling and data sharing in late-binding overlays,” in Proceedings of the International Conference on High Performance Computing & Simulation (HPCS). IEEE, 2014, pp. 129–136.

[8] D. S. Katz, S. Jha, M. Parashar, O. Rana, and J. Weissman, “Survey and analysis of production distributed computing infrastructures,” arXiv preprint arXiv:1208.2649, 2012.

[9] C. Sehgal, “Opportunistic eco-system & OSG-XD update,” Presentation at OSG Council Meeting, April 11, 2014, https://indico.fnal.gov/event/8389/contributions/107137/attachments/70027/83972/OSG-XD_Report_to_Council_11apr2014.pdf

[10] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Wurthwein ¨ et al., “The open science grid,” in Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC) conference, Journal of Physics: Conference Series, vol. 78(1). IOP Publishing, 2007, p. 012057.

[11] Open Science Grid (OSG), http://www.opensciencegrid.org/ (дата обращения: 23.11.2022)

[12] K. De, A. Klimentov, T. Wenaus, T. Maeno, and P. Nilsson, “PanDA: A new paradigm for distributed computing in HEP through the lens of ATLAS and other experiments,” ATL-COM-SOFT-2014-027, Tech. Rep., 2014.

[13] G. Aad, E. Abat, J. Abdallah, A. Abdelalim, A. Abdesselam, O. Abdinov, B. Abi, M. Abolins, H. Abramowicz, E. Acerbi et al., “The ATLAS experiment at the CERN large hadron collider,” Journal of Instrumentation, vol. 3, no. 08, p. S08003, 2008.

[14] LHC Study Group, “The large hadron collider, conceptual design,” CERN/AC/95-05 (LHC) Geneva, Tech. Rep., 1995.

[15] D. Bonacorsi, T. Ferrari et al., “WLCG service challenges and tiered architecture in the LHC era,” in IFAE 2006. Springer, 2007, pp. 365–368.

[16] Worldwide LHC Computing Grid (WLCG), The Apache software foundation, http://wlcg.web.cern.ch/ (дата обращения: 15.02.2023)

[17] G. Juve, “The Glidein service,” Presentation, http://www.slideserve.com/embed/5100433

[18] I. Sfiligoi, “glideinWMS—a generic pilot-based workload management system,” in Proceedings of the international conference on computing in high energy and nuclear physics (CHEP2007), Journal of Physics: Conference Series, vol. 119(6). IOP Publishing, 2008, p. 062044.

[19] M. Hategan, J. Wozniak, and K. Maheshwari, “Coasters: uniform resource provisioning and access for clouds and grids,” in Proceedings of the 4th IEEE International Conference on Utility and Cloud Computing (UCC). IEEE, 2011, pp. 114–121.

[20] J. T. Mo´scicki, “DIANE - distributed analysis environment for GRID-enabled simulation and analysis of physics data,” in Proceedings of the IEEE Nuclear Science Symposium Conference Record, vol. 3. IEEE, 2003, pp. 1617–1620.

[21] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev et al., “DIRAC pilot framework and the DIRAC Workload Management System,” in Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series, vol. 219(6). IOP Publishing, 2010, p. 062049.

[22] P.-H. Chiu and M. Potekhin, “Pilot factory – a Condor-based system for scalable Pilot Job generation in the Panda WMS framework,” in Proceedings of the 17th International Conference on Computing in High Energy and Nuclear Physics (CHEP09), Journal of Physics: Conference Series, vol. 219(6). IOP Publishing, 2010, p. 062041.

[23] A. Rubio-Montero, E. Huedo, F. Castej´on, and R. Mayo-Garc´ıa, “GWpilot: Enabling multi-level scheduling in distributed infrastructures with gridway and pilot jobs,” Future Generation Computer Systems, vol. 45, pp. 25–52, 2015.

[24] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid,” in Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, vol. 1. IEEE, 2000, pp. 283–289.

[25] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, “Falkon: a Fast and Light-weight tasK executiON framework,” in Proceedings of the 8th ACM/IEEE conference on Supercomputing. ACM, 2007, p. 43.

[26] E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, “Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment,” in Proceedings of the IEEE Challenges of Large Applications in Distributed Environments (CLADE) workshop. IEEE, 2006, pp. 95–103.

[27] PanDAWMS – pilot3 [Электронный ресурс] // URL: https://github.com/PanDAWMS/pilot3/wiki (дата обращения: 28.10.2022)

[28] DIRAC pilot framework and the DIRAC Workload Management System [Электронный ресурс] // URL: https://iopscience.iop.org/article/10.1088/1742-6596/219/6/062049/pdf (дата обращения: 16.03.2023)

[29] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees, “DIRAC: A scalable lightweight architecture for high throughput computing,” in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. IEEE Computer Society, 2004, pp. 19–25.

[30] J. Kim, W. Huang, S. Maddineni, F. Aboul-Ela, and S. Jha, “Exploring the RNA folding energy landscape using scalable distributed cyberinfrastructure,” in Emerging Computational Methods in the Life Sciences, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ser. HPDC ’10. New York, NY, USA: ACM, 2010, pp. 477–488.

[31] B. K. Radak, M. Romanus, T.-S. Lee, H. Chen, M. Huang, A. Treikalis, V. Balasubramanian, S. Jha, and D. M. York, "Characterization of the Three-Dimensional Free Energy Manifold for the Uracil Ribonucleoside from Asynchronous Replica Exchange Simulations," Journal of Chemical Theory and Computation, vol. 11, no. 2, pp. 373–377, 2015, http://dx.doi.org/10.1021/ct500776j

[32] A. Merzky, O. Weidner, and S. Jha, "SAGA: A standardized access layer to heterogeneous distributed computing infrastructure " Software-X, 2015, dOI: http://dx.doi.org/10.1016/j.softx.2015.03.001

[33] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers," 2015.

Приложения

```
class JobData(BaseData):
    """
    High-level object to host Job definition/settings
    """

    jobid = None                # unique Job identifier (forced to be a string)
    scriptname = ""               # script execution name
    taskid = None               # unique Task identifier, the task that this job belongs to (forced to be a string)
    jobparams = ""               # job parameters defining the execution of the job
    # current job status; format = {key: value, ..} e.g. key='LOG_TRANSFER', value='DONE'
    status = Running
    platform = ""               # CPU/CPU+GPU
    # set by the pilot (not from job definition)
    workdir = ""                # working directory for this job
```

Листинг 1. Реализация класса с параметрами задачи.

```
def run(args):
    """
    Main execution function for the generic workflow.

    The function sets up the internal queues which handle the flow of jobs.

    :param args: pilot arguments.
    :returns: traces.
    """

    logger.info('setting up queues')
    queues = namedtuple('queues', ['jobs', 'payloads', 'data_in', 'data_out','validated_jobs', 'validated_payloads',
'monitored_payloads', 'finished_jobs', 'finished_payloads', 'failed_jobs', 'failed_payloads', 'failed_data_in',
'failed_data_out', 'completed_jobs'])

    queues.jobs = queue.Queue()
    queues.payloads = queue.Queue()
    queues.data_in = queue.Queue()
    queues.data_out = queue.Queue()

    queues.validated_jobs = queue.Queue()
    queues.validated_payloads = queue.Queue()
```

```
queues.monitored_payloads = queue.Queue()

queues.finished_jobs = queue.Queue()
queues.finished_payloads = queue.Queue()
queues.finished_data_in = queue.Queue()
queues.finished_data_out = queue.Queue()

queues.failed_jobs = queue.Queue()
queues.failed_payloads = queue.Queue()
queues.failed_data_in = queue.Queue()
queues.failed_data_out = queue.Queue()

queues.completed_jobs = queue.Queue()

# queues.interceptor_messages = queue.Queue()

logger.info('setting up tracing')
traces = namedtuple('traces', ['pilot'])
traces.pilot = {'state': SUCCESS,
                'error_code': 0,
                'command': None}

# define the threads
targets = {'job': job.control, 'payload': payload.control, 'data': data.control, 'monitor': monitor.control}
threads = [ExcThread(bucket=queue.Queue(), target=target, kwargs={'queues': queues, 'traces': traces, 'args': args},
                     name=name) for name, target in list(targets.items())]

logger.info('starting threads')
[thread.start() for thread in threads]

logger.info('waiting for interrupts')

# the thread_count is the total number of threads, not just the ExcThreads above
thread_count = threading.activeCount()
abort = False
while threading.activeCount() > 1 or not abort:
    for thread in threads:
        bucket = thread.get_bucket()
        try:
            exc = bucket.get(block=False)
        except queue.Empty:
            pass
        else:
            print(f'received exception from bucket queue in generic workflow: {exc_obj}', file=stderr)
```

```
        thread.join(0.1)

    abort = threads_aborted()
    if abort:
        logger.debug(f'all relevant threads have aborted (thread count={threading.activeCount()})')
        break

    sleep(1)

logger.info(f'end of generic workflow (traces error code: {traces.pilot["error_code"]})')

return traces
```

Листинг 2. Код запуска потоков.