

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ.....	4
ВВЕДЕНИЕ	6
ГЛАВА 1. ОБЗОР.....	8
1.1. ОПИСАНИЕ СРЕДЫ.....	8
1.2. АРХИТЕКТУРА CRIU	9
1.3. ОГРАНИЧЕНИЯ CRIU	12
1.4. ПОСТАНОВКА ЗАДАЧИ.....	14
Выводы по главе 1	14
ГЛАВА 2. ТЕОРЕТИЧЕСКОЕ ИССЛЕДОВАНИЕ	16
2.1. CRIU И ПЛАТФОРМЫ КОНТЕЙНЕРИЗАЦИИ.....	16
2.2. ПРОБЛЕМА С ТСР СОЕДИНЕНИЕМ	17
2.3. ПРОБЛЕМА С ИЗМЕНЯЮЩЕЙСЯ ФАЙЛОВОЙ СИСТЕМОЙ.....	18
2.3.1. Описание проблемы	18
2.3.2. Решение проблемы	19
2.4. ПРОБЛЕМА С ЗАПУСКОМ JUPYTER LAB ПОСЛЕ ПЕРЕНОСА	20
2.5. ПРОБЛЕМА С ПЕРЕНОСОМ УСТАНОВЛЕННЫХ ПОЛЬЗОВАТЕЛЕМ	
БИБЛИОТЕК	20
Выводы по главе 2	22
ГЛАВА 3. ПРАКТИЧЕСКОЕ ИССЛЕДОВАНИЕ	23
3.1. СРЕДА ВНЕДРЕНИЯ И ВЗАИМОДЕЙСТВИЕ С ВЫЧИСЛИТЕЛЬНЫМИ	
МАШИНАМИ.....	23
3.2. ДОБАВЛЕНИЕ В DOCKER-PY ВЗАИМОДЕЙСТВИЯ С CRIU	25
3.2.1. Сущность Checkpoint	26
3.2.2. Сущность Checkpoint Collection.....	26
3.2.3. Метод Docker start	27
3.2.4. Метод container checkpoints.....	27
3.2.5. Метод container checkpoint create.....	27
3.2.6. Метод container remove checkpoint	28

3.3. СЕРВИС СОЗДАНИЯ И ВОССТАНОВЛЕНИЯ КОНТРОЛЬНЫХ ТОЧЕК	28
3.3.1. Создание контрольной точки	28
3.3.2. Восстановление контрольной точки	29
3.3.3. Взаимодействие пользователя с контрольными точками ...	30
3.3.4. Расширение Jupyter lab	31
3.4. СРАВНЕНИЕ С АНАЛОГАМИ	33
ВЫВОДЫ ПО ГЛАВЕ 3	37
ЗАКЛЮЧЕНИЕ	39
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	40

ВВЕДЕНИЕ

В последнее время все большую популярность приобретает использование языка программирования Python для решения различных задач, связанных с веб-разработкой, анализом данных, машинным обучением, научными вычислениями и автоматизацией. Python является языком высокого уровня с открытым исходным кодом, который характеризуется удобочитаемостью кода, простотой и удобством использования, и большим количеством разработанных библиотек. Python это интерпретируемый язык. Интерпретатор Python считывает исходный код построчно и выполняет его в режиме реального времени, что делает его популярным выбором для создания прототипов, тестирования и быстрой разработки [4].

У разработчиков в области машинного обучения и анализа данных одним из основных применяемых инструментов является Jupyter lab – популярное веб приложение с открытым исходным кодом, которое позволяет пользователям работать с Jupyter notebooks, текстовыми редакторами, терминалами и настраиваемыми компонентами гибким, интегрированным и расширяемым образом. Jupyter notebooks – интерактивная среда, позволяющая пользователям исполнять ячейки с кодом или текстом. Одним из основных языков программирования, использующимся в качестве ядра Jupyter lab является Python.

В машинном обучении, для обучения моделей, требуются дорогостоящие вычисления с использованием GPU – одного из видов микропроцессоров. В GPU содержатся несколько сотен вычислительных ядер, с помощью которых сложные расчеты выполняются очень быстро. Часто создание моделей машинного обучения можно разбить на этапы, такие как:

1. Сбор данных;
2. Подготовка данных;
3. Обучение модели на подготовленных данных;
4. Использование обученной модели.

Сбор и подготовка данных могут быть достаточно долгими по времени, и при этом не требовать дорогостоящих вычислительных ресурсов. С целью

уменьшения стоимости хотелось бы иметь возможность применять для этого процесса дешевые ресурсы, а для процесса обучения модели переносить полученное состояние на более дорогостоящую машину, чтобы использовать ее эффективно. Сохранение состояния и его восстановление позволяет пользователям возобновить их работу, без необходимости повторного запуска кода или перезагрузки данных, что экономит вычислительные ресурсы и время разработки.

Основным подходом для переноса состояния Jupyter lab является сериализация – процесс перевода структуры данных в битовую последовательность. Подход с сериализацией подразумевает сохранение состояния каждой сущности по отдельности [3].

CRIU (Checkpoint/Restore In Userspace) – программное обеспечение, позволяющее создавать снапшот состояния контейнера или отдельного приложения.

Цель работы: реализовать перенос состояния Python интерпретатора между вычислительными узлами с использованием CRIU.

Для достижения цели необходимо решить следующие задачи:

1. Реализовать перенос состояния Python интерпретатора, используя CRIU;
2. Провести сравнительный анализ с существующими решениями;
3. Внедрить полученное решение.

Для удобства при проведении исследования будет переноситься состояние ядра Jupyter lab.

В результате выполнения работы требуется получить способ переноса состояния Python интерпретатора с помощью CRIU, который позволит сохранять и восстанавливать пользовательский код с меньшим количеством ограничений, чем сериализация.

ГЛАВА 1. ОБЗОР

1.1. ОПИСАНИЕ СРЕДЫ

Jupyter lab – среда разработки, которая позволяет работать пользователям с Jupyter notebook, поддерживающим исполнение кода, написанного на различных языках программирования, при этом обычно в Jupyter notebook исполняют код, созданный на языке программирования Python. Jupyter lab используют специалисты в области машинного обучения и анализа данных. Так же существуют различные облачные сервисы, предоставляющие возможности работы с Jupyter notebook или Jupyter lab, иногда с переработанным интерфейсом, такие как: Google Colab, Yandex Datasphere или JetBrains Datalore.

Ядро Jupyter lab по умолчанию не сохраняет свое состояние, поэтому при его перезагрузке необходимо по новой исполнять код [8]. Это приводит к дополнительным тратам вычислительных ресурсов и времени разработки. При этом стоимость вычислений на машинах, например, с использованием GPU, может быть критична, и имеет смысл использовать их по максимуму, исполняя только ту часть кода, где это действительно необходимо, а те вычисления, для которых не нужен GPU (для сбора и обработки данных) производить на более дешевых вычислительных машинах.

Перенос состояния ядра Jupyter lab подразумевает под собой перенос состояний переменных, установленных библиотек, и других объектов, загруженных во время исполнения Jupyter notebook.

Так же, например, сервис Yandex Datasphere [5], реализует serverless режим для Jupyter lab. В нем необходимо поддерживать возможность исполнения кода пользователей на различных вычислительных конфигурациях, с возможностью их выбора при исполнении каждой ячейки. При этом, при переходе на другую вычислительную машину состояние интерпретатора Python должно переноситься между вычислительными узлами и вычисления должны продолжаться с того места, на котором они остановились до сохранения состояния.

Например, рассмотрим следующие этапы разработки моделей машинного обучения:

1. Сбор и обработка данных;
2. Обучение модели;
3. Использование модели.

Выполнение этапов, связанных со сбором и обработкой данных, а также использованием обученной модели может быть не столь требовательно к вычислительным ресурсам, как этап обучения модели. Serverless режим и возможность переноса состояния Python интерпретатора с одной вычислительной машины на другую, способствуют значительной экономии пользовательских средств.

Основным способом решения задачи переноса состояния интерпретатора является сериализация. Это процесс перевода структуры данных в битовую последовательность. Подход с сериализацией подразумевает сохранение состояния каждой сущности по отдельности, что делает процессы сохранения и восстановления состояния сильно зависимыми от объектов, которые необходимо сериализовать. Некоторые из объектов невозможно сериализовать, например, файловые дескрипторы или tcp соединения, и при этом для большого количества объектов может не поддерживаться их сериализация в библиотеке по умолчанию, что будет требовать для каждого такого объекта необходимость реализовывать отдельные сериализаторы. При этом подходе у пользователей часто возникают проблемы с тем, что для объектов, которые они хотят использовать не добавлены сериализаторы, вследствие чего сохранить и перенести состояние процесса полностью не предоставляется возможным.

1.2. АРХИТЕКТУРА CRIU

CRIU (Checkpoint/Restore In Userspace) – программное обеспечение, позволяющее заморозить запущенный контейнер или отдельное приложение и создать контрольную точку его состояния. Сохраненные данные могут быть перенесены на другую вычислительную машину, восстановлены, и приложение

или контейнер продолжит свою работу с места, в котором контейнер был заморожен. Использовать эту функциональность можно для создания снапшотов, удаленной отладки приложений и прочего [2].

Всю информацию, относящуюся к процессу, для которого необходимо создать контрольную точку, CRIU сохраняет в одном или нескольких файлах изображения. Эти файлы изображения содержат информацию о процессе, такую как:

- страницы памяти;
- дескрипторы файлов;
- межпроцессное взаимодействие;
- и так далее.

Для успешной реализации создания контрольной точки или её восстановления CRIU реализует как можно больше функциональности в пользовательском интерфейсе [6].

Одним из наиболее важных интерфейсов ядра для CRIU является интерфейс ptrace. Ptrace — системный вызов в некоторых unix-подобных системах, который позволяет трассировать или отлаживать выбранный процесс. CRIU полагается на возможность отслеживать процесс с помощью ptrace. Затем он вводит «паразитный» код для сброса страниц памяти процесса в файлы изображения из адресного пространства процесса. Под адресным пространством понимается множество допустимых адресов объектов вычислительной системы, к которым относятся ячейки памяти, сектора диска, узлы сети. Данные адреса могут использоваться для доступа к этим объектам при определенном режиме работы.

Для каждой части процесса, для которого создается контрольная точка, создаются отдельные файлы изображения. Например, информация о страницах памяти собирается из следующих частей:

- /proc/\$PID/smmaps;
- /proc/\$PID/mapfiles;

- /proc/\$PID/pagemap.

Файлы изображения страниц памяти требуют наибольшего объема памяти, особенно по сравнению с остальными файлами изображениями, которые содержат дополнительную информацию о процессе, для которого создается контрольная точка, такую как:

- открытые файлы;
- credentials;
- регистры;
- состояние задачи;
- и так далее.

Чтобы создать контрольную точку для дерева процесса, т.е. процесса и всех его дочерних процессов, CRIU создает контрольную точку у каждого подключенного дочернего процесса.

Для восстановления процесса из контрольной точки, CRIU использует информацию, собранную во время ее создания. Восстановить процесс можно только если у него тот же PID (идентификатор процесса), который был у него при создании контрольной точки. Если же другой процесс использует этот PID, восстановление закончится с ошибкой. Одна из причин, по которой процесс должен быть восстановлен с тем же PID, заключается в том, что деревья родительски-дочерних процессов должны быть восстановлены точно такими, какими они были. Чтобы восстановить процесс с тем же PID, используется интерфейс ядра, позволяющий влиять на то, какой PID ядро предоставит новому процессу.

Если процесс, созданный с помощью инструкции clone, имеет корректный PID, CRIU преобразует его в то же состояние, в котором процесс находился до создания контрольной точки. Файлы открываются и располагаются также, как они были раньше, память восстанавливается в том же состоянии, а вся остальная информация из файлов изображений используется для восстановления процесса. Как только состояние восстановлено, оставшиеся части системы восстановления процесса удаляются. Затем восстановленный процесс

возобновляет управление и продолжается с того состояния, в котором он был при создании контрольной точки.

1.3. ОГРАНИЧЕНИЯ CRIU

У CRIU есть ряд ограничений, одним из главных, является то, что CRIU может использовать межпроцессную связь для создания и восстановления контрольных точек процесса, только если процессы выполняются внутри межпроцессного пространства имен. Существующие отношения родитель-потомок в деревьях процессов должны быть сохранены нетронутыми. Это означает, что CRIU при создании контрольной точки и ее восстановлении использует родительский процесс и все его дочерние процессы. Невозможно создать контрольную точку и перезапустить родительский процесс самостоятельно. Это ограничение связано с требованием, чтобы PID оставался неизменным. Процесс восстановления CRIU завершается неудачей, если предполагаемый PID уже используется в системе. Для успешной миграции, используемые библиотеки должны быть точно такой же версии, как в исходной, так и в целевой системах. Это ограничение существует потому, что процесс уже загрузил все необходимые библиотеки и ожидает, что функции, предоставляемые этими библиотеками, находятся по тому же адресу, что и при запуске.

У процессов, содержащих некоторые объекты могут быть созданы контрольные точки при специальных опциях, к таким объектам относятся:

- Внешние ресурсы. По умолчанию CRIU позволяет создавать контрольные точки у наборов процессов и их ресурсов в случае, если у этого набора нет подключений извне. В некоторых ситуациях имеет смысл игнорировать внешнее соединение при создании контрольной точки и ее восстановлении.

- Блокировки файлов. Блокировка файла – это объект, который принадлежит некоторой файловой системе. При создании контрольной точки невозможно выяснить, может ли эта блокировка, помогающая одной задаче,

использоваться какой-то другой задачей. Таким образом, CRIU не создает контрольные точки задачи с удержанными блокировками, но при этом опция `filelocks`, указывает CRIU создавать контрольные точки блокировок.

- Невидимые файлы. Иногда имя файла не может быть найдено в файловой системе. В этом случае CRIU может оставить для него временное название.

При этом, для некоторых объектов не может быть создана контрольная точка. К ним относятся:

- Устройства. Если задача открыла или сопоставила какое-либо символьное или блочное устройство, это обычно означает, что ей требуется какое-то подключение к оборудованию. В этом случае создание контрольной точки и ее восстановление невозможны. Исключение составляют виртуальные устройства, такие как `null`, `zero` и т.д., а также сетевое устройство `TUN`, которое используется `Open VPN`.

- Открытые файлы из размонтированной файловой системы. Когда занятая файловая система "лениво" размонтирована, любые ссылки на нее очищаются, как только файловая система больше не занята. Если у процесса есть открытый файл из файловой системы с отложенным размонтированием, CRIU просто не может создать контрольную точку у этого процесса и восстановить её, если только файл не закрыт.

- Задачи с подключенным отладчиком. CRIU использует тот же API, что и отладчики, для получения состояния некоторых задач, и этот API (`ptrace one`) не позволяет нескольким отладчикам исследовать задачу. Таким образом, у задач, использующих отладчики по типу `gdb` или `strace` не могут быть созданы контрольные точки.

- Задача от другого пользователя (для некорневых пользователей). По соображениям безопасности если CRIU запрашивается некорневым пользователем для создания контрольной точки какой-либо задачи от другого пользователя, CRIU не выполняет этого, если получатель контрольной точки не принадлежит тому же пользователю.

- Сокеты, отличные от TCP, UDP, UNIX, packet и netlink.
- Каналы, созданные с помощью O_DIRECT.
- UDP сокеты, использующие опцию UDP_CORK.
- Сегмент памяти SysVIPC без пространства имен IPC. Объекты IPC не привязаны ни к каким задачам. Таким образом, как только CRIU встречает память IPC, подключенную к задаче, требуется также сброс всего пространства имен IPC.
- Графические приложения. Создание контрольной точки и её восстановление у приложения подключенного к "реальному" Xserver (например, на ноутбуке) невозможно из-за того, что часть состояния приложения находится на Xserver, и CRIU не создает контрольную точку.

1.4. ПОСТАНОВКА ЗАДАЧИ

Для решения обозначенных ранее проблем и достижения цели работы, необходимо выполнить следующие действия:

1. Реализовать перенос состояния Python интерпретатора, используя CRIU;
2. Провести сравнительный анализ предлагаемого способа с существующими решениями;
3. Внедрить полученное решение.

Реализация переноса состояния Python интерпретатора, используя CRIU включает в себя:

- Исследование области применения CRIU и ограничений в его работе;
- Подготовку окружения для работы CRIU;
- Реализацию сервисов, осуществляющих перенос состояния.

При осуществлении сравнительного анализа, необходимо оценить области применения реализованного способа, с альтернативами.

ВЫВОДЫ ПО ГЛАВЕ 1

В этой главе рассмотрено:

- Среда, в которой будет реализоваться работа;

- Проблемы, с которыми сталкиваются специалисты в области машинного обучения и анализа данных;
- Проблемы возникающие при переносе состояния способом использующим сериализацию;
- Возможности CRIU, способствующие решению проблем.

Так же, в этой главе проведено исследование области применения CRIU и ограничений в его работе. Описано восстановление процесса из контрольной точки, с применением CRIU, использующим информацию, собранную во время ее создания. А также приведена постановка задачи.

ГЛАВА 2. ТЕОРЕТИЧЕСКОЕ ИССЛЕДОВАНИЕ

2.1. CRIU и ПЛАТФОРМЫ КОНТЕЙНЕРИЗАЦИИ

Для реализации задачи переноса состояния, как было описано выше, будет использован CRIU.

Существуют различные платформы контейнеризации. Контейнеризация – технология, которая позволяет запускать приложения в пространствах, изолированных на уровне операционной системы (ОС). Контейнеры являются наиболее распространенной формой виртуализации на уровне ОС.

Docker – это платформа контейнеризации, позволяющая разработчикам упаковывать приложения и их зависимости в контейнеры, которые затем можно последовательно запускать в различных средах. Docker упрощает последовательную упаковку и развертывание приложений в различных средах, снижая риск возникновения проблем с совместимостью. Docker также обладает высокой степенью масштабируемости и может быть легко интегрирован с другими инструментами оркестровки контейнеров, например Kubernetes.

CRIU может использоваться для переноса контейнеров, в частности CRIU интегрирован в OpenVZ, LXC/LXD, Docker, Podman и другие приложения. Использование контейнеров для нашей задачи позволяет помимо состояния ядра Jupyter lab, переносить так же и состояние расширений и терминалов, которые могут быть важны пользователю. Помимо этого, использование контейнеров позволит упростить перенос состояния, ведь CRIU при создании контрольной точки, помимо страниц памяти, файловых дескрипторов и прочего, автоматически будет сохранять файлы, относящиеся к приложению, работающему в контейнере.

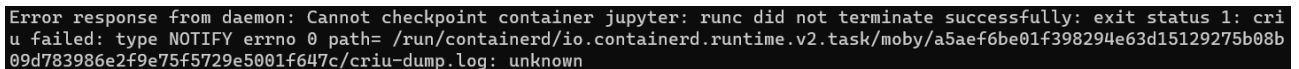
CRIU переносит контейнер или приложение целиком, что позволяет сильно увеличить количество объектов возможных для сохранения и восстановления, вследствие чего этот способ может значительно улучшить пользовательский опыт.

В существующей инфраструктуре Yandex Datasphere, для которой разрабатывается перенос состояния, ядро, на котором выполняется пользовательский код, запущенно в Docker контейнере. Это добавляет ограничения для решения поставленных задач.

CRIU интегрирован в Docker лишь частично, и работает только в экспериментальном режиме Docker. Разработчики CRIU не стремятся дополнять функционал доступный для использования в Docker. При этом они рекомендуют использовать другую платформу контейнеризации для работы с CRIU, например Podman, но в рамках решения задачи, как было описано выше, требуется использовать именно Docker. В следующих параграфах будут описаны проблемы, связанные с этим.

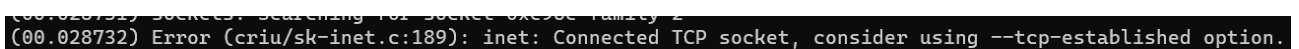
2.2. ПРОБЛЕМА С ТСП СОЕДИНЕНИЕМ

По умолчанию при попытке создать контрольную точку у веб-приложения, которым является Jupyter lab, возникает ошибка, показанная на рисунке 1. В файлах логов (рис. 2) в ошибке сказано рассмотреть возможность использования опции `--tcp-established`.



```
Error response from daemon: Cannot checkpoint container jupyter: runc did not terminate successfully: exit status 1: criu failed: type NOTIFY errno 0 path= /run/containerd/io.containerd.runtime.v2.task/moby/a5aef6be01f398294e63d15129275b08b09d783986e2f9e75f5729e5001f647c/criu-dump.log: unknown
```

Рис. 1. Ошибка при создании контрольной точки



```
(00.028732) sockets: searching for socket object family 2
(00.028732) Error (criu/sk-inet.c:189): inet: Connected TCP socket, consider using --tcp-established option.
```

Рис. 2. Ошибка в логах

Существует две опции способные помочь в решении ошибки, описанной выше. Опция `tcp-established` и опция `tcp-close`. Опция `tcp-established` используется для того, чтобы явно указать, что вызывающая сторона знает о «переходном» состоянии сетевого фильтра. Опция `tcp-close` позволяет слушающим ТСП-сокетам существовать после создания/восстановления контрольной точки дерева процессов. Она сбрасывает все остальные ТСП-соединения в дереве процессов. Все клиенты сервера смогут повторно подключиться к сокетам, созданным во время восстановления дерева процессов. В случае задачи переноса веб-приложения был сделан вывод, что лучше использовать опцию `tcp-close`. Она так

же позволяет переносить приложение в среду с другими IP-адресами, при условии, что сокеты прослушивают адрес 0.0.0.0, что не получится сделать при использовании tcp-established. После установки опции tcp-close, контрольная точка успешно создается.

2.3. ПРОБЛЕМА С ИЗМЕНЯЮЩЕЙСЯ ФАЙЛОВОЙ СИСТЕМОЙ

2.3.1. Описание проблемы

Для восстановления контрольной точки на вычислительной машине должен быть создан Docker образ до запуска восстановления. При этом сам контейнер, должен быть остановлен, восстановление контрольной точки во время работы контейнера не приводит ни к каким изменениям состояния контейнера. На прошлом шаге была установлена опция tcp-close и создана контрольная точка. При попытке восстановления из нее, процесс зависает, в логах Docker выводится сообщение, показанное на рисунке 3. При просмотре логов работы CRIU, можно увидеть ошибку отсутствия файлов (рис. 4)

```
May 20 20:33:08 fmyar-2 dockerd[13396]: time="2023-05-20T20:33:08.530299163Z" level=error msg="stream copy error: reading from a closed fifo"
May 20 20:33:08 fmyar-2 dockerd[13396]: time="2023-05-20T20:33:08.534847702Z" level=error msg="stream copy error: reading from a closed fifo"
```

Рис. 3. Фрагмент из логов, ошибка при восстановлении контейнера

```
(00.280463) 95: Error (criu/files-reg.c:2259): Can't open file home/jovyan/.ipython/profile_default/history.sqlite on restore: No such file or
directory
(00.280472) 95: Error (criu/files-reg.c:2185): Can't open file home/jovyan/.ipython/profile_default/history.sqlite: No such file or directory
```

Рис. 4. Фрагмент из логов, ошибка отсутствия файлов

Вышеописанная проблема происходит по тому, что CRIU в Docker не поддерживает создание контрольных точек и их восстановление при изменениях в файловой системе, это приводит к ошибкам отсутствия файлов при восстановлении контрольной точки, пример показан на рисунке 5 (в строках логов с номерами 2835 и 2839).

```
2834 (00.196887) 79: sockets: restore mark 0 for socket
2835 (00.196889) 7: Error (criu/files-reg.c:2259): Can't open file home/jovyan/.local/share/jupyter/nbsignatures.db on restore: No such fil
2836 (00.196891) 79: sockets: restore priority 0 for socket
2837 (00.196894) 79: sockets: restore rcvlowat 1 for socket
2838 (00.196896) 79: sockets: restore mark 0 for socket
2839 (00.196897) 7: Error (criu/files-reg.c:2185): Can't open file home/jovyan/.local/share/jupyter/nbsignatures.db: No such file or direct
2840 (00.196898) 79: sockets: set keepcnt for socket
```

Рис. 5. Фрагмент из логов, ошибки при создании контрольной точки

2.3.2. Решение проблемы

Для решения этой проблемы необходимо переносить состояние файловой системы, с которой взаимодействует пользователь отдельно, без использования CRIU. Один из вариантов сохранения состояния файловой системы: создание Docker volume на директории, с которыми взаимодействует пользователь. Docker volumes – рекомендуемый разработчиками Docker способ хранения данных. В Linux docker volumes находятся по умолчанию в `/var/lib/docker/volumes/`. Другие программы не должны получать к ним доступ напрямую, только через контейнер. В документации к Docker представлена схема работы Docker volume [1], она изображена на рисунке 6.

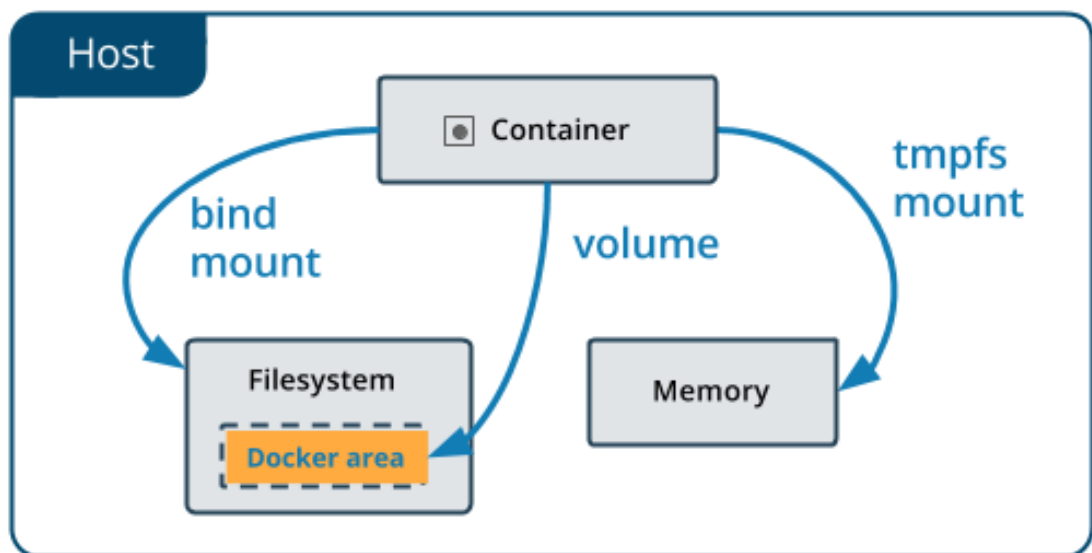


Рис. 6. Схема Docker volume

При использовании Docker volumes CRIU не сохраняет состояние файлов в них и не добавляет их в контрольную точку, вследствие чего эти файлы необходимо переносить отдельно. При этом для восстановления контейнера при применении Docker volumes, контейнер должен быть создан с необходимыми volumes до момента запуска с контрольной точки, а после чего остановлен, для возможности восстановления состояния.

Для переноса состояния Jupyter lab, необходимо создать volume на домашнюю директорию внутри контейнера, например, для Docker образа `jupyter/scipy-notebook` требуется volume на `/home/jovyan/`, являющуюся

домашней директорией в образе `jupyter/scipy-notebook`. В ней хранятся файлы, с которыми работает пользователь и необходимые для работы Jupyter lab.

2.4. ПРОБЛЕМА С ЗАПУСКОМ JUPYTER LAB ПОСЛЕ ПЕРЕНОСА

После запуска контейнера с добавленным `volume` на домашнюю директорию, необходимо установить права доступа к этой директории. Далее контейнер успешно стартует, ячейки выполняются. После переноса состояния контейнер успешно восстанавливается из контрольной точки, но при запуске интерфейса Jupyter lab и попытке открыть ноутбук, возникает ошибка, показанная на рисунке 7, и в логах Jupyter lab можно увидеть следующее: `sqlite3.OperationalError: attempt to write a readonly database`. Ошибка возникает из-за того, что после переноса внутренняя `sqlite` база данных `nbsignatures` доступна для записи только текущему пользователю.

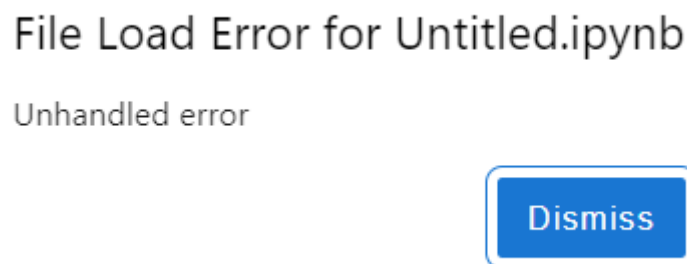


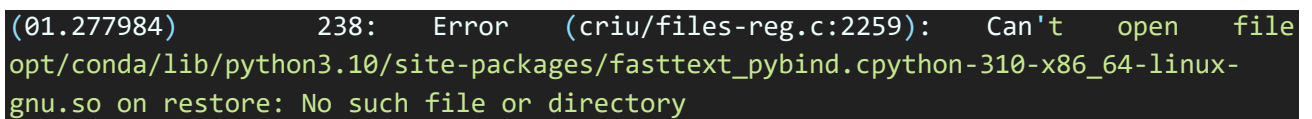
Рис. 7. Ошибка при открытии ноутбука

Решить эту проблему удалось добавив опция запуска Jupyter lab: `NotebookNotary.db_file=:memory:`, которая отключает запись на диск `sqlite` базы данных. В общем случае это может привести к тому, что после перезапуска контейнера данные будут потеряны, но в условиях задачи состояние этой базы данных будет сохранено в контрольной точке с помощью `CRIU`, и при восстановлении из нее данные не будут потеряны.

2.5. ПРОБЛЕМА С ПЕРЕНОСОМ УСТАНОВЛЕННЫХ ПОЛЬЗОВАТЕЛЕМ БИБЛИОТЕК

Попробуем установить библиотеку через `pip` и перенести состояние, после этого.

Рассмотрим пример с установкой библиотеки `fastText`. `FastText` - библиотека для эффективного изучения представлений слов и классификации предложений. Установим ее через `pip` и подключим через `import`, и обучим модель на тестовой выборке. Попробуем перенести состояние `Jupyter lab` после обучения. Контрольная точка успешно создается, но при ее восстановлении после переноса `docker start` зависает. В логах `journalctl docker` можно увидеть ошибку: `stream copy error: reading from a closed fifo`. В логах восстановления `CRIU` можно увидеть ошибку, показанную на рисунке 8.



```
(01.277984)      238:  Error   (criu/files-reg.c:2259):  Can't open file
opt/conda/lib/python3.10/site-packages/fasttext_pybind.cpython-310-x86_64-linux-
gnu.so on restore: No such file or directory
```

Рис. 8. Ошибка при восстановлении состояния с установленной библиотекой `fastText`

Ошибка возникает по той же причине, что и при переносе состояния без добавленного `volume` на домашнюю директорию. Для решения этой проблемы необходимо пользовательские библиотеки переносить отдельно, не полагаясь на `CRIU`. Для этого укажем `pip` устанавливать пакеты в отдельную директорию, добавив в конфиг `pip` опцию `target` на эту директорию, например `/pips`. Так же необходимо контейнеру при запуске указать переменную окружения `PYTHONPATH` на `/pips`, и предоставить права на нее. `PYTHONPATH` - переменная среды, которая используется для указания расположения библиотек `Python`. После этого перенос состояния выполнится успешно.

Так же, по причине того, что пользовательский код может удерживать блокировки на файлы, если попытаться создать контрольную точку, то возникает ошибка. Как было описано выше в этом случае необходимо указать опцию `file-locks` для того, чтобы `CRIU` создавал контрольную точку блокировки, взятой на файл.

ВЫВОДЫ ПО ГЛАВЕ 2

В этой главе были рассмотрены преимущества использования систем контейнеризации в рамках решаемой задачи. Описаны проблемы, возникающие при создании контрольной точки и ее восстановлении и их решения:

- Проблема с tcp соединением
- Проблема с

ГЛАВА 3. ПРАКТИЧЕСКОЕ ИССЛЕДОВАНИЕ

3.1. СРЕДА ВНЕДРЕНИЯ И ВЗАИМОДЕЙСТВИЕ С ВЫЧИСЛИТЕЛЬНЫМИ МАШИНАМИ

Для реализации serverless режима Jupyter lab в Yandex Datasphere, разделяют машины, на которых запущены контейнеры с интерфейсом Jupyter lab, и контейнеры с ядром Jupyter lab, исполняющим пользовательский код. Схема использования изображена на рисунке 9. При этом, при выборе dedicated режима, можно применять такую же схему взаимодействия, но пользователю полностью в пользование выделяется Kernel на время его работы в проекте.

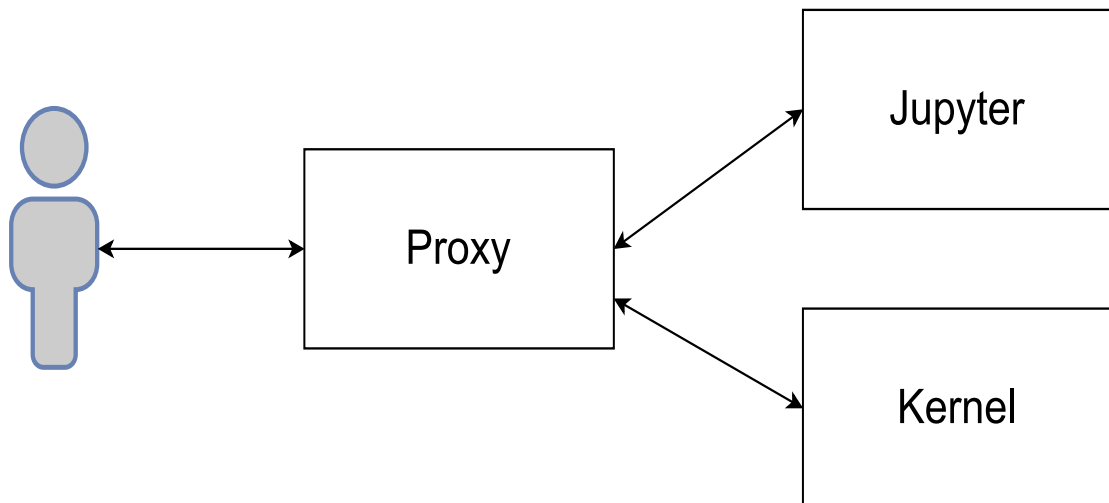


Рис. 9. Схема использования Jupyter lab пользователями

При создании проекта, для пользователя создается вычислительная машина с запущенным Proxy сервисом, и машина с интерфейсом Jupyter notebook с предустановленными расширениями, необходимыми для работы. Proxy создает WebSocket соединение с Jupyter notebook и с пользователем, обрабатывая запросы. Перед запуском ячеек, выделяется Kernel – вычислительная машина, которая используется для исполнения пользовательского кода. В serverless режиме, существует возможность выбора вычислительной машины перед исполнением каждой ячейки. В dedicated режиме, пользователю выделяется Kernel сразу при запуске проекта, и должна быть реализована возможность переноса состояния, в момент необходимый пользователю.

Yandex Datasphere предоставляет возможность эксплуатации и serverless режима и dedicated режима. На рисунке 10 показана схема использования Yandex Datasphere пользователем.

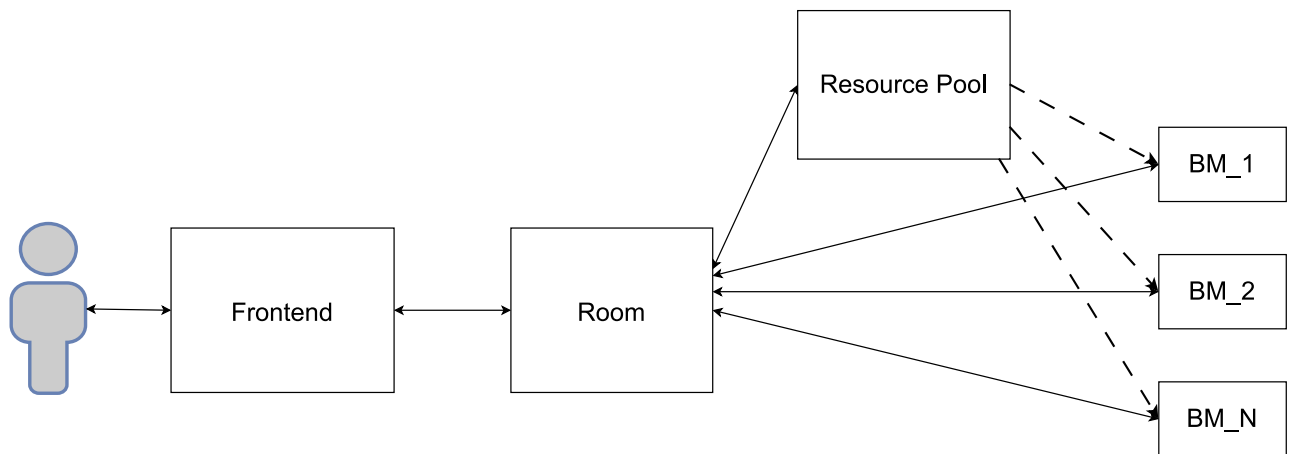


Рис. 10. Схема использования в Yandex Datasphere

Room – сервис, исполняющий роль Proху из схемы, представленной на рисунке 8. Resource Pool – сервис отвечающий за создание и управление вычислительными машинами, при этом поддерживающий для каждой конфигурации набор вычислительных машин с возможностью быстрого предоставления их пользователю. Когда пользователю необходимо исполнить код, Room отправляет запрос в Resource Pool на предоставление ресурсов, и в последствии реализует вышеописанный процесс.

Схема взаимодействия с вычислительными машинами в Yandex Datasphere до внедрения переноса состояния с использованием CRIU показана на рисунке 11. На каждой вычислительной машине запущен контейнер, либо ядро, на котором исполняется пользовательский код, либо интерфейс Jupyter lab. При этом, когда необходимо создать контрольную точку, Proху отправляет запрос на Kernel и получает состояние переменных. После этого загружает их в S3. S3 (Simple Storage Service) – это сервис хранения объектов, имеющий высокие показатели производительности, доступности и безопасности данных [7].

А при восстановлении на Kernel, где уже запущен контейнер, отправляет на него состояние переменных и Kernel загружает их с помощью десериализации.

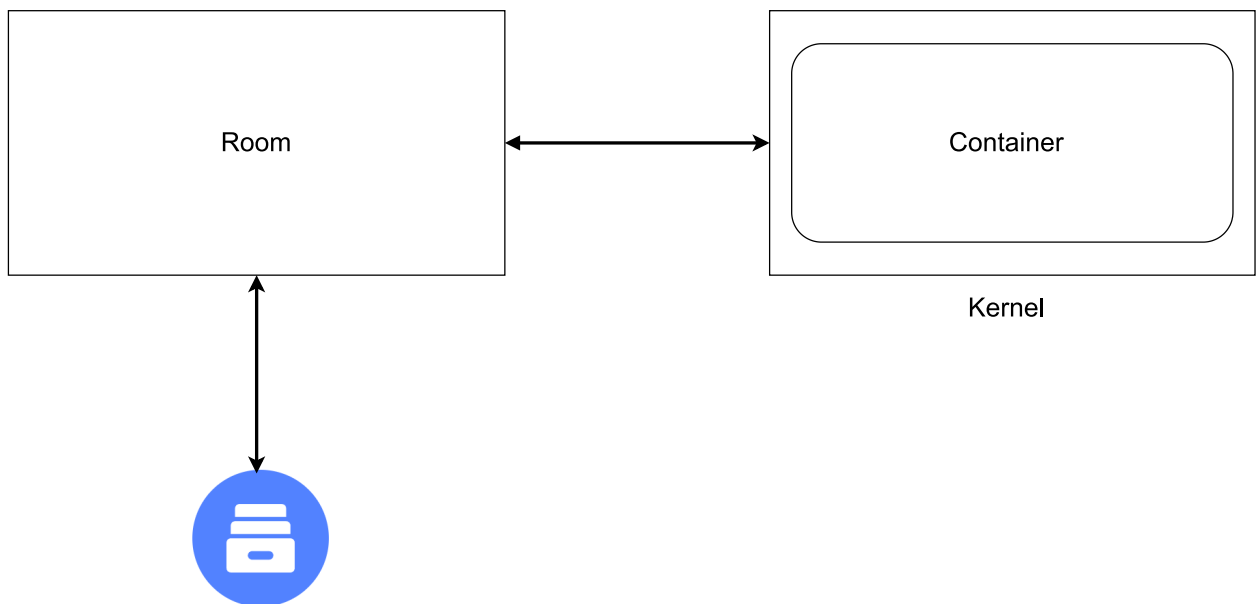


Рис. 11. Схема взаимодействия с VM

3.2. ДОБАВЛЕНИЕ В DOCKER-PY ВЗАИМОДЕЙСТВИЯ С CRIU

При решении задачи переноса состояния, была переработана схема создания контрольной точки. Разработан Python сервис [9] запускаемый параллельно с Docker контейнером на вычислительной машине, это показано на рисунке 12. В Python service реализована возможность создания и загрузки состояния, для этого используется доработанный docker-py для создания контрольной точки и запуска контейнера с нее. Docker-py –это библиотека с открытым исходным кодом, предоставляющая возможность работы с Docker, напрямую из Python приложения. В процессе выполнения работы она была расширена для взаимодействия с Docker checkpoints. Далее описаны методы и сущности добавленные в Docker-py.

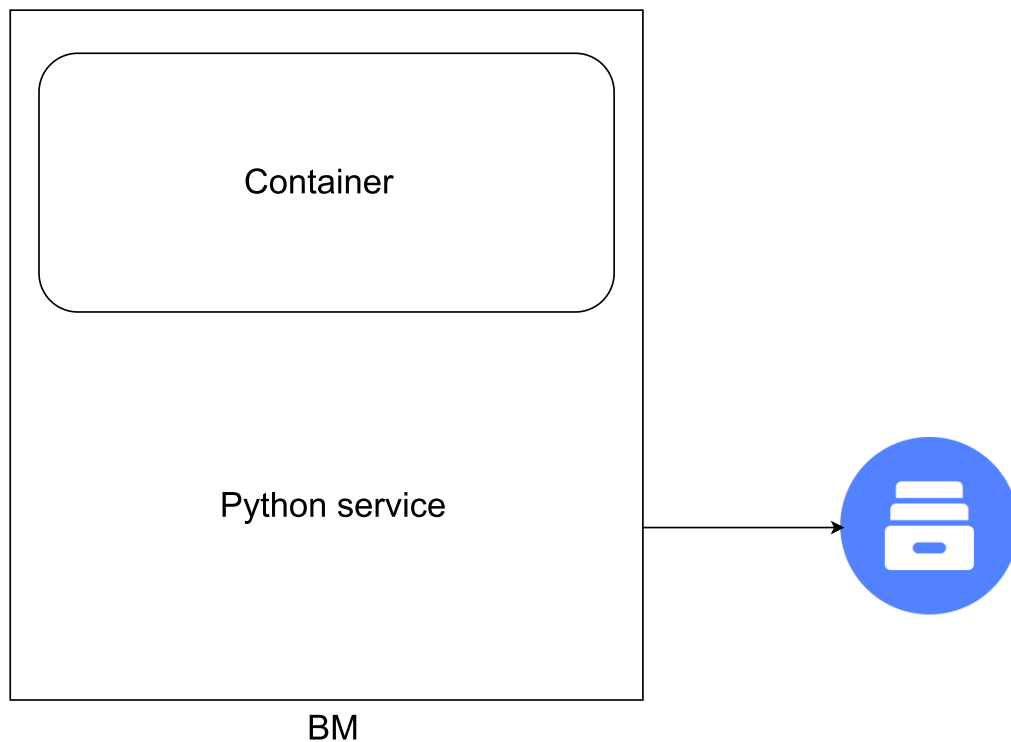


Рис. 12. Сервисы, запущенные на вычислительной машине

3.2.1. Сущность Checkpoint

Разработана сущность Checkpoint, унаследованная от Model, базового класса в библиотеке Docker-py, от которого должны наследоваться все сущности. Эта сущность возвращается в результате выполнения методов, которые будут рассмотрены далее.

3.2.2. Сущность Checkpoint Collection

Разработана сущность Checkpoint Collection, унаследованная от Collection, базового класса в библиотеке Docker-py, от которого должны наследоваться все коллекции сущностей. При создании коллекции контрольных точек необходимо указать id контейнера и опционально checkpoint_dir – директорию, с которой будет взаимодействовать коллекция. В Checkpoint Collection были разработаны методы, позволяющие помимо методов базового класса:

- Создавать новую контрольную точку, в директории в которой находится коллекция;

- Искать контрольную точку по имени среди контрольных точек коллекции;
- Получать список всех контрольных точек в коллекции;
- Очищать все контрольные точки коллекции.

Так же был разработан метод `get_checkpoints`, использующийся для получения Checkpoint Collection у контейнера.

3.2.3. Метод Docker start

Доработан метод `start`, который запускает уже созданный контейнер. Для восстановления контрольной точки, в Docker API в метод `start`, нужно передать имя контрольной точки, и опционально директорию, в которой она находится, по умолчанию, это директория `/var/lib/docker/containers/{container_id}/checkpoints`. В Docker-ру доработан метод `start` в которой было добавлено два опциональных параметра: `checkpoint` и `checkpoint_dir`. В этом методе выполняется проверка на их существование, и после они добавляются в параметры `post` запроса, который отправляется в Docker API. При этом параметр `checkpoint` может принимать на вход, как и описанную выше сущность Checkpoint, так и имя контрольной точки.

3.2.4. Метод container checkpoints

Разработан метод `container checkpoints` – получение списка контрольных точек у контейнера. Этот метод является аналогом CLI команды `docker checkpoint list`. Метод принимает на вход `id` контейнера, у которого необходимо получить список контрольных точек и опциональный параметр `checkpoint_dir` – директорию, в которой находятся контрольные точки, добавленные в результат метода. После этого посылается `get` запрос в Docker API, и возвращается список контрольных точек.

3.2.5. Метод container checkpoint create

Разработан метод `container checkpoint create` – создание контрольной точки у контейнера, принимающего `id` контейнера, имя контрольной точки, и опционально `checkpoint_dir` – директорию, в которой будет создана контрольная

точка и опционально `leave_running` – будет ли остановлен контейнер после создания контрольной точки. `Leave_running` используется для реализации `live migration` – переноса виртуальной машины с одного физического сервера на другой без прекращения работы виртуальной машины и остановки сервисов. После обработки формируется тело `post` запроса, и он отправляется в `Docker API`.

3.2.6. Метод `container remove checkpoint`

Разработан метод `container remove checkpoint` – удаление контрольной точки у контейнера. На вход метод принимает два обязательных параметра: `id` контейнера и имя контрольной точки, а также опциональный параметр `checkpoint_dir` – директорию, из которой должна быть удалена контрольная точка. После обработки параметров формируется `delete` запрос, и он отправляется в `Docker API`.

3.3. СЕРВИС СОЗДАНИЯ И ВОССТАНОВЛЕНИЯ КОНТРОЛЬНЫХ ТОЧЕК

3.3.1. Создание контрольной точки

`Python service` принимает запросы на вход двух роутеров. Первый роутер – метод `save_state`. Схема его работы продемонстрирована на рисунке 13. Этот метод предназначен для сохранения состояния контейнера. На вход принимается имя контрольной точки и `leave running` параметр, который будет передан в `Docker-ru`. После обработки параметров, необходимо сохранить состояние файловой системы, с которой работает пользователь – те `Docker volumes`, которые были созданы в главе 2, сохранить их в архив и загрузить в `S3` хранилище. После загрузки `Docker volumes`, с помощью доработанного `Docker-ru` создается контрольная точка, и загружается после создания в `S3`.

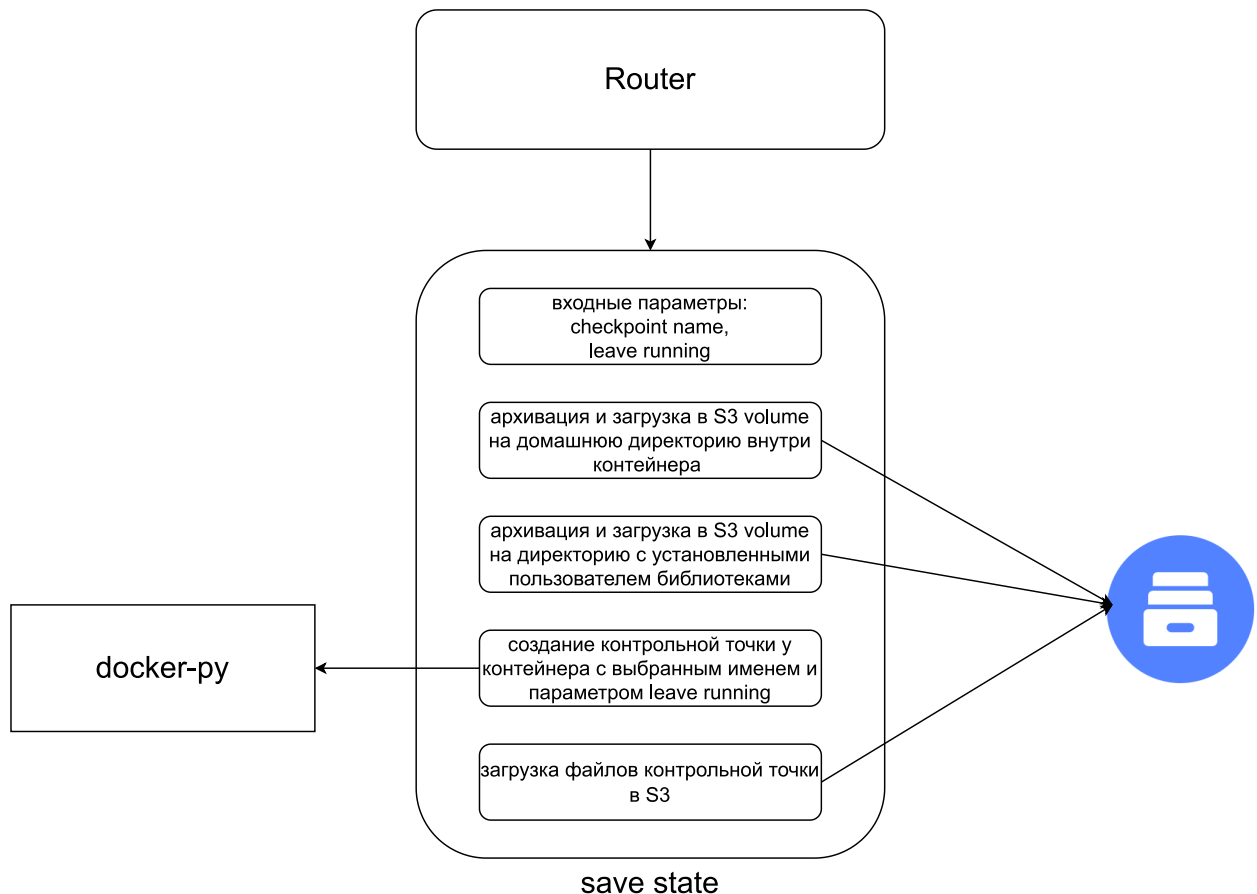


Рис. 13. Код метода save_state

3.3.2. Восстановление контрольной точки

Второй роутер – метод load_state. Схема его работы показана на рисунке 14. Этот метод предназначен для восстановления контрольной точки. Перед восстановлением контрольной точки, если контейнер запущен, он останавливается, это нужно для того, чтобы загрузить состояния на уже использующуюся вычислительную машину. В случае загрузки состояния на вычислительную машину, еще не используемую пользователем, контейнер будет создан, но выключен. Из S3 загружаются файлы необходимые для восстановления Docker volumes и сама контрольная точка, которую необходимо восстановить, загруженные в методе save_state. После загрузки архивы разархивируются в необходимое место на диске. И с помощью доработанного метода в Docker-py – docker start, загружается контрольная точка.

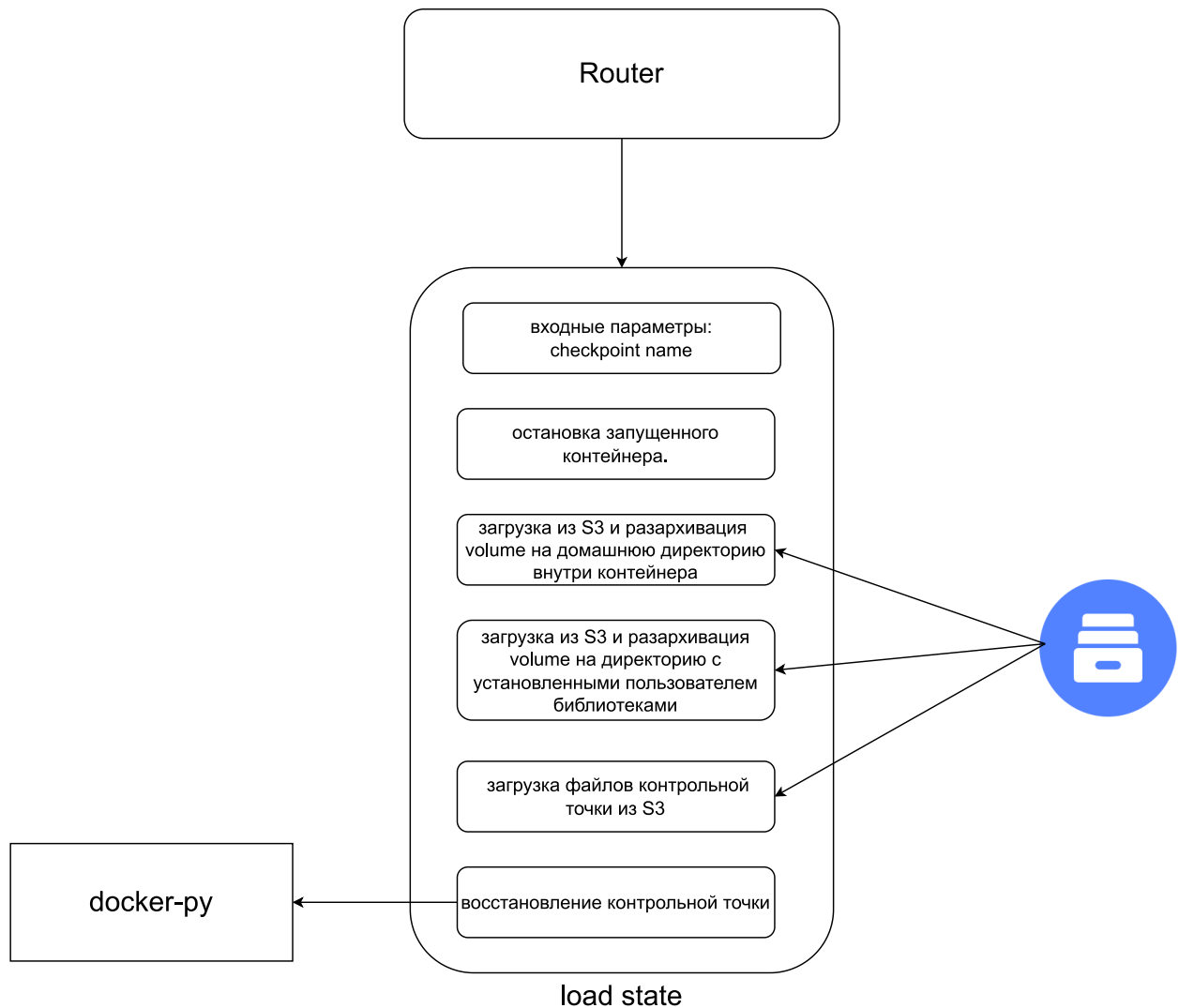


Рис. 14. Код метода load_state

3.3.3. Взаимодействие пользователя с контрольными точками

Как было описано выше, в Kernel и Jupyter запущены Docker контейнеры с соответствующими образами. С целью переноса состояния Python интерпретатора, требуется создавать контрольные точки у Kernel, используя CRIU. Пользователю необходимо предоставить возможность создавать контрольные точки для сохранения состояния, без его переноса. В этом помогает live migration, и опциональный параметр при создании контрольной точки – leave running. Так же пользователю предоставляется возможность возобновить свою работу с созданной контрольной точки. При ручной остановке проекта, смены вычислительной машины или отсутствия действий на ней необходимо создавать

контрольную точку. При создании контрольной точки в этих случаях, можно выключать контейнер автоматически, не указывая опцию `leave running`.

При использовании CRIU появляется возможность применить его к интерфейсу Jupyter lab запущенному на вычислительной машине – Jupyter, тем самым сохранив его состояние, и позволить пользователю устанавливать расширения, так же сохраняя их состояния, и поддерживать список установленных расширений. Для этого необходимо создавать контрольные точки контейнера Jupyter lab. Но при этом, состояние Jupyter lab сохраняется только в случае выключения проекта или отсутствия действий пользователя в нем, при котором останавливается контейнер с Jupyter lab. В отличии от Jupyter lab, состояние контейнера Kernel переносится чаще во время работы пользователя и используется для переноса состояния объектов, созданных во время исполнения кода.

3.3.4. Расширение Jupyter lab

Было разработано расширение для Jupyter lab, позволяющее пользователю выбирать вычислительную машину, на которой будет исполняться код. На рисунке 15 показано окно выбора вычислительной машины. На рисунке 16 изображено окно со списком вычислительных машин, предоставленных пользователю для выбора. Пользователь выбирает вычислительную машину из списка и нажимает кнопку ОК, после чего на Проху отправляется запрос на смену вычислительной машины. Проху обрабатывает запрос, и вызывает `save_state` у Python service на вычислительной машине, выделенной пользователю на момент запроса. После создания контрольной точки, отправляется запрос `load_state` на выбранную пользователем вычислительную машину, и после его выполнения, Проху переподключает web socket соединение и запросы пользователя идут на новую вычислительную машину.

Switch to vm

Cancel

OK

Рис. 15. Окно выбора вычислительной машины

Switch to vm

VM-3	▼
VM-3	
VM-2	
VM-1	

Рис. 16. Список предоставленных пользователю вычислительных машин

Как было упомянуто выше, для восстановления контрольной точки требуется, чтобы на вычислительной машине, на момент восстановления, был создан образ с подключенными volumes, но при этом контейнер должен быть остановлен. По этой причине, при создании пользователем нового проекта, чтобы не останавливать уже запущенный контейнер, была создана контрольная точка, соответствующая стартовому состоянию. Из нее восстанавливается состояние для новых проектов.

Пользователю в интерфейсе доступен список созданных контрольных точек, из каждой контрольной точки предоставляется возможность восстановления состояния и продолжения работы с этим состоянием. Список контрольных точек продемонстрирован на рисунке 17.

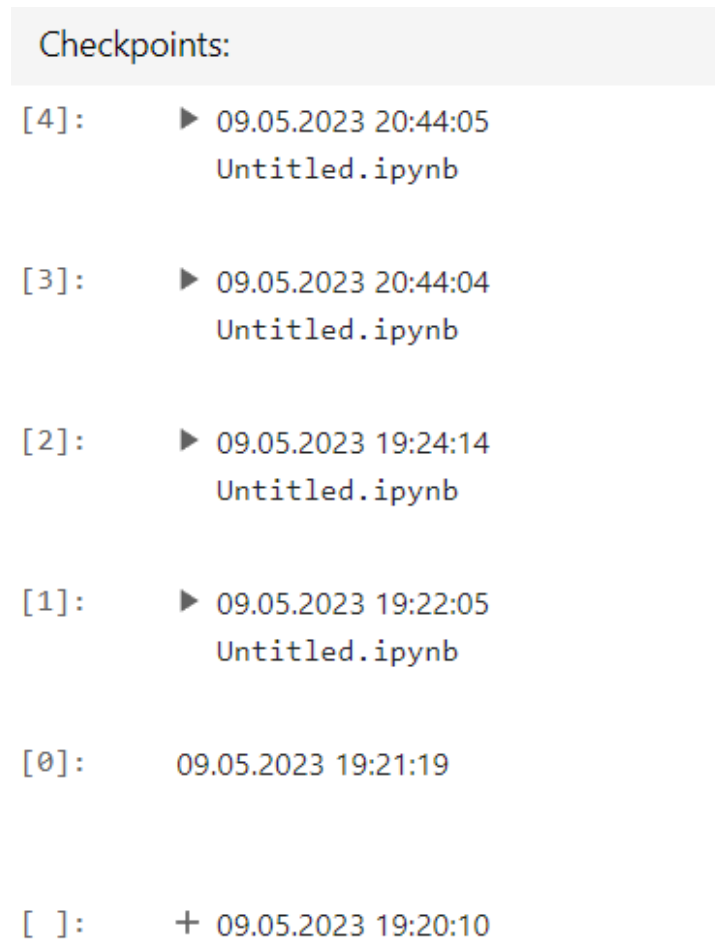


Рисунок 17. Список контрольных точек доступных пользователю

Таким образом, в результате проделанной работы, был реализован перенос состояния Python интерпретатора с помощью CRIU, который решает проблемы, возникающие у специалистов в сфере машинного обучения и анализа данных. Создан удобный пользовательский интерфейс, которые позволяет пользователю осуществлять перенос простым нажатием кнопок и выбором вычислительной машины, на которую будет осуществлен перенос.

3.4. СРАВНЕНИЕ С АНАЛОГАМИ

Сравним реализованное решение с основным способом переноса состояния, использующим сериализацию.

Сравнение по области применения. В следующих пунктах показано в каких случаях CRIU успешно справляется с переносом состояний, а с помощью сериализации невозможно это сделать:

- Файловые дескрипторы. На рисунке 18 показан фрагмент кода читающий данные из файла. В файле записаны подряд идущие числа от 1 до 6. После исполнения кода (рис. 19) происходит перенос состояния, и выполняется чтение из файла, как видно на рисунке 19 данные продолжают считываться с места, в котором была создана контрольная точка. При этом файловые дескрипторы нельзя сериализовать полностью и достичь вышеописанного поведения.

```
f = open('file.txt', 'r')
```

```
f.readline()
```

```
'1\n'
```

```
f.readline()
```

```
'2\n'
```

Рис. 18. Считывание данных из файла до переноса

```
f.readline()
```

```
'3\n'
```

```
f.readline()
```

```
'4\n'
```

Рис. 19. Продолжение считывания данных из файла после переноса

- Grpc канал. На рисунке 20 показан код, исполненный на вычислительной машине до переноса состояния. Сначала создается grpc канал с удаленным хостом. После чего создается stub на этом канале, и посылается запрос на сервер, который возвращает в результате число. На рисунке 21 показан код, исполненный после переноса состояния. Как можно увидеть, grpc канал успешно переносится, и позволяет без пересоздания посылать запросы. Сериализовывать grpc каналы, невозможно, CRIU же позволяет решить перенос grpc каналов.

```
[30]: import grpc
import bidirectional_pb2_grpc
import bidirectional_pb2

[31]: host = "158.160.25.120"
channel = grpc.insecure_channel(target=f"{host}:50051")
stub = bidirectional_pb2_grpc.ChatServerStub(channel)

[32]: def gen_note(i):
note = bidirectional_pb2.Note()
note.name = f'test{i}'
note.message = f'test{i}'
return note

[35]: r = stub.SendNote(gen_note(2))
r

[35]: res: 2
```

Рис. 20. Код создания grpc канала и отправка запроса по grpc до переноса состояния

```
[36]: r = stub.SendNote(gen_note(2))
r

[36]: res: 3

[37]: r = stub.SendNote(gen_note(2))
r

[37]: res: 4
```

Рис. 21. Отправка запросов после переноса состояния без пересоздания grpc канала

- **Sqlite база данных.** На рисунке 22 показан запуск ноутбука с подключенным в memory режиме sqlite до переноса состояния. В sqlite загружается датасет, после чего исполняется запрос на получение всех записей в базе данных, получая результаты по одному. На рисунке 23 показано продолжение получения данных из базы данных, после переноса состояния. Было перенесено и состояние базы данных, и состояние cursor, вследствие чего получение результатов продолжается с места, в котором была создана контрольная точка. Сериализаторов для sqlite базы данных не написано.

```
[1]: import sqlite3
import pandas

[2]: conn = sqlite3.connect(':memory:')
df = pandas.read_csv('accessories.csv')

[3]: r = df.to_sql('usage', conn, if_exists='replace', index=False)

[4]: cursor = conn.cursor()
cursor.execute('select * from usage')
cursor.fetchone()

[4]: ('3D glasses',
      'White',
      'No',
      '490',
      122,
      'White',
      'Colorful',
      '1x1',
      None,
      'Able Sisters',
      'Available from Able Sisters shop only',
      'All Year',
      'No',
      '1.0.0',
      'Active',
      'party',
      'AccessoryEye',
      'Yes',
      'For sale',
      'AccessoryGlassThreed0',
      4463,
      'FNxEraBTwRiCvtFu')

[5]: cursor.fetchone()

[5]: ('3D glasses',
      'Black',
      'No',
      '490',
      122,
      'Black',
      'Colorful',
      '1x1',
      None,
      'Able Sisters',
      'Available from Able Sisters shop only',
      'All Year',
      'No',
      '1.0.0',
      'Active',
      'party',
      'AccessoryEye',
      'Yes',
      'For sale',
      'AccessoryGlassThreed0',
      4463,
      'FNxEraBTwRiCvtFu')
```

Рис 22. Код получения данных из sqlite базы данных до переноса состояния

```
[6]: cursor.fetchone()

[6]: ('beak',
      'Yellow',
      'No',
      '490',
      122,
      'Yellow',
      'Yellow',
      '1x1',
      None,
      'Able Sisters',
      "Available from either Mable's temporary shop or Able Sisters shop",
      'All Year',
      'Yes',
      '1.0.0',
      'Cute',
      'fairy tale; party; theatrical',
      'AccessoryMouthInvisibleNose',
      'No',
      'For sale',
```

Рис. 23. Получение данных из sqlite после переноса состояния

- Fasttext модель. По умолчанию в библиотеках с реализациями сериализации, например, cloudpickle, не реализовано сохранение модели Fasttext. Для этого необходимо регистрировать сериализатор, написанный авторами библиотеки. Реализация, представленная в работе, при этом, переносит модель без проблем.
- Toloka-kit. Для нее не написаны сериализаторы, поэтому подход с сериализацией не позволяет перенести, например Pool, наша реализация с CRIU переносит его без проблем.

Из вышеописанного видно, что по области применения реализованный способ с применением CRIU выигрывает у подхода с сериализацией.

Сравнительный анализ показал, что полученное решение позволяет переносить большее количество пользовательского кода, при этом перенос состояния не зависит от подключенных сериализаторов и возможности сериализации объектов в целом.

ВЫВОДЫ ПО ГЛАВЕ 3

В результате проделанной работы, был реализован перенос состояния Python интерпретатора с помощью CRIU, который решает проблемы,

возникающие у специалистов в сфере машинного обучения и анализа данных с помощью простого и понятного пользовательского интерфейса.

Сравнительный анализ показал, что полученное решение позволяет переносить большее количество пользовательского кода, при этом перенос состояния не зависит от подключенных сериализаторов и возможности сериализации объектов в целом.

ЗАКЛЮЧЕНИЕ

В ходе проделанной работы в рамках исследования были рассмотрены:

- Среда, в которой реализовано исследование;
- Проблемы, с которыми сталкиваются специалисты в области машинного обучения и анализа данных и способы их решения;
- Проблемы, возникающие при переносе состояния способом, использующим сериализацию;
- Возможности CRIU, способствующие решению проблем;

Так же было проведено исследование области применения CRIU и ограничений в его работе. Кроме того, в работе описаны преимущества использования систем контейнеризации.

В результате проделанной работы, был реализован перенос состояния Python интерпретатора с помощью CRIU, который решает проблемы, возникающие у специалистов в сфере машинного обучения и анализа данных.

Сравнительный анализ, разработанного способа переноса с существующими, показал, что полученное решение позволяет переносить большее количество пользовательского кода, при этом перенос состояния не зависит от подключенных сериализаторов и возможности сериализации объектов в целом.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Документация Docker volume [Электронный ресурс] – Режим доступа: <https://docs.docker.com/storage/volumes/> (дата обращения: 19.05.2023)
2. Документация CRIU [Электронный ресурс] – Режим доступа: https://criu.org/Main_Page (дата обращения 19.05.2023)
3. Документация Jupiter Lab [Электронный ресурс] – Режим доступа: <https://docs.jupyter.org/en/latest/> (дата обращения 19.05.2023)
4. Документация Python [Электронный ресурс] – Режим доступа: <https://www.python.org/> (дата обращения 19.05.2023)
5. Yandex Datasphere [Электронный ресурс] – Режим доступа: <https://datasphere.yandex.ru/> (дата обращения 19.05.2023)
6. Архитектура CRIU [Электронный ресурс] – Режим доступа: <https://access.redhat.com/articles/2455211> (дата обращения 19.05.2023)
7. Документация Amazon S3 [Электронный ресурс] – Режим доступа: <https://aws.amazon.com/ru/s3/getting-started/> (дата обращения 19.05.2023)
8. Jupyter Lab extensions [Электронный ресурс] – Режим доступа: https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html (дата обращения 19.05.2023)
9. Документация Flask [Электронный ресурс] – Режим доступа: <https://flask.palletsprojects.com/en/2.3.x/> (дата обращения 19.05.2023)
10. Документация Java [Электронный ресурс] – Режим доступа: <https://docs.oracle.com/en/java/javase/17/> (дата обращения 19.05.2023)
11. Документация Spring [Электронный ресурс] – Режим доступа: <https://docs.spring.io/spring-framework/reference/> (дата обращения 19.05.2023)