Санкт–Петербургский государственный университет

ТЕРЕЩЕНКО Дмитрий Владиславович

Выпускная квалификационная работа

Отказоустойчивые распределенные вычисления на Python на основе управляющих объектов

Уровень образования: бакалавриат
Направление 02.03.02 «Фундаментальная информатика и информационные технологи»
Основная образовательная программа «Программирование и информационные технологи» №17/5003/1
Профиль «Автоматизация научных исследований»

Научный руководитель:
доцент, Кафедра компьютерного моделирования и многопроцессорных систем, к.ф. – м.н. Ганкевич Иван Геннадьевич

Рецензент:
аналитик, Дирекция Инновационного Развития, Центр Разработки и Монетизации Данных, ООО «Газпромнефть – Цифровые Решения»
Миронов Евгений Геннадьевич

Санкт-Петербург
2021

Saint–Petersburg State University

TERESHCHENKO Dmitrii Vladislavovich

Graduation project

Fault-tolerant distributed computing in Python based on control flow objects

Level of education: Bachelor level
Main field of study 02.03.02 «Fundamental Informatics and Information Technology»
Academic programme title «Programming and Information technology»
№17/5003/1
Area of specialisation (if applicable): «Automation of Scientific Researches»

Scientific supervisor:
Candidate of Physics and Mathematics
I.G.Gankevich

Reviewer:
analyst, Innovation Development Directorate,
Data Development and Monetization Center,
LLC «Gazprom Neft Digital Solutions»
E.G.Mironov

Saint–Petersburg
2021

Содержание

Термины и сокращения

Термин, сокращение	**Толкование**
Байт-код	стандартное промежуточное представление, в которое может быть переведена компьютерная программа автоматическими средствами
Виртуальная машина	программа, которая эмулирует компьютерную систему в изоляции от другого программного обеспечения на одной вычислительной системе
Дамп памяти	содержимое рабочей памяти одного процесса, ядра или всей операционной системы
Декоратор	функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода
Интерпретатор	программа (разновидность транслятора), выполняющая интерпретацию
Интерпретация	построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения)
Интроспекция	возможность запросить тип и структуру объекта во время выполнения программы
Контрольная точка	служебная информация, которая записывается операционной системой на жесткий диск или другой носитель для обработки исключительных ситуаций, таких как перезапуск, сбой или отказ оборудования

Кластер	группа компьютеров, объединённых высокоскоростными каналами связи, представляющая с точки зрения пользователя единый аппаратный ресурс
НИОКР	научно-исследовательские и опытно-конструкторские работы
Сборщик мусора	одна из форм автоматического управления памятью
Скрипт (сценарий)	это последовательность действий, описанных с помощью скриптового языка программирования
Сериализация	процесс перевода структуры данных в последовательность байтов. Обратной к операции сериализации является операция десериализации
Узел	устройство (типовой компьютер), соединённое с другими устройствами как часть компьютерной сети
Фреймворк	программная платформа, определяющая структуру программной системы
Файл SGY	файл данных, сохраненный в формате SEG-Y (общество геологоразведочных геофизиков). Он содержит геофизические данные в двоичном и текстовом формате, который включает в себя координаты отраженных сейсмических волн
Файл NPY	стандартный бинарный файл для хранения одного произвольного NumPy-массива на диске. Формат хранит всю необходимую информацию о форме и типе данных для правильного восстановления массива даже на другой машине с другой архитектурой
gRPC	система удалённого вызова процедур
GIL	Global Interpreter Lock (Способ синхронизации потоков)

Введение

Экосистема языка программирования Python включает в себя множество инструментов для обработки и анализа данных, за счёт чего он становится всё больше востребован в области научных вычислений. Но сегодня реальность такова, что всё чаще появляются задачи, когда вычислительных ресурсов одного компьютера не хватает. В таком случае прибегают к распределенным вычислениям, т.е. вычислениями, производимым на распределенной вычислительной системе.

Любая распределённая вычислительная система представляет из себя множество индивидуальных сущностей (узлов) (см. рис. 1), для связи которых используются различные специализированные программные интерфейсы.

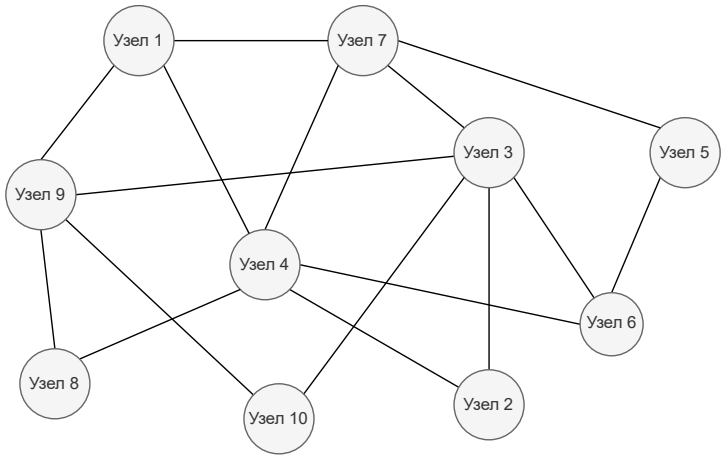


Рис. 1: Иллюстративный пример распределённой вычислительной системы.

Для такого рода систем создание универсального интерфейса, который бы позволял программировать их без непосредственной привязки к количеству узлов, их физическим характеристикам, количеству параллельных процессов, сетевым адресам и т.п., упирается в свойственную им ненадежность. Действительно, при рассмотрении единственного вычислительного узла, его выход из строя является критической ситуацией, которую нельзя разрешить на уровне программного обеспечения, из-за чего все программы, предназначенные для запуска на одном узле, написаны в предположении, что этот узел абсолютно надежен. Для распределенной системой

достаточно большого размера ситуация обстоит совершенно иначе. Выход из строя одного из узлов в данном случае является нормальной ситуацией, поэтому данный интерфейс в таком случае должен предоставлять механизмы для сохранения работоспособности системы в целом. На текущий день в распределенных системах используются достаточно примитивные механизмы обеспечения отказоустойчивости: контрольные точки, дампы памяти и другие интенсивные операции ввода-вывода.

Таким образом, текущие программные интерфейсы, как в целом, так и в Python, либо слишком узкоспециализированные, либо не дают полной отказоустойчивости.

Обзор существующих решений

На текущий день для построения распределенных вычислений в Python существует несколько различных решений [1]. Каждое из них отличается друг от друга тем, как и на чём построена модель распределённых вычислений, а также целями, ради которых оно используется.

Одно из самых популярных применений Python – машинной обучение. Для этой области характерна ситуация, когда настройка параметров модели выливается в достаточно тяжеловесную задачу, в связи с чем и возник фреймворк **Ray** [2, 3], основанный на модели акторов [4] с поддержкой общего состояния и gRPC коммуникаций.

В Ray весь жизненный цикл акторов и их метаданные (например, IP-адрес и порт) управляются службой GCS (Global Control Store) [5], расположенной на головном узле. Рабочие узлы Ray спроектированы так, чтобы быть однородными, так что любой отдельный узел может быть потерян без разрушения всего кластера. Текущим исключением из этого правила является головной узел, так как на нем размещается GCS. В настоящее время предпринимаются усилия по поддержке высокой доступности для GCS, чтобы он мог работать на любом узле, а также распределяться на несколько узлов. Таким образом, на текущий момент Ray не даёт полной отказоустойчивости.

Помимо машинного обучения Python также широко используется в науке. Однако для использования в этой области у него есть несколько ограничений, в первую очередь связанных с распараллеливанием и масштабируемостью. **Parsl** [6, 7], библиотека параллельного программирования с открытым исходным кодом для Python, направлена на удовлетворение этих потребностей.

Parsl включает в себя две новые конструкции. Первая – это приложение, обозначаемое декоратором вокруг функции Python, которая может выполняться асинхронно, и которая объявляет информацию о зависимости ввода и вывода для связывания с другими приложениями. Это обеспечивает интуитивный, неявный и естественный способ выражения параллелизма. При вызове приложения Parsl возвращает вторую новую конструкцию,

объект, называемый фьючерсом [4].

Программа Parsl может выйти из строя из-за сбоя одного из ее приложений или узла, используемого для выполнения. По мере увеличения размеров анализа возрастает и вероятность неудачи. Чтобы Parsl можно было использовать, он должен ожидать сбоев и реагировать соответствующим образом. Например, при повторном исполнении ветви неудачного выполнения пользователь вряд ли захочет повторно запустить другую ветвь, которая успешно завершилась. Parsl обеспечивает отказоустойчивость на уровне всей программы, а не приложений. Это гарантирует, что при желании пользователь может повторно запустить программу, и любые приложения, вызываемые с теми же аргументами, не должны быть повторно выполнены.

В независимости от рассматриваемой области часто встречается задача обработки данных. Обычно для этих целей используется также Python за счёт большого количества инструментов, входящих в его экосистему. В случае, когда объём данных начинает возрастать, часто приходят к парадигме параллельных вычислений MapReduce [8], поэтому в Python возник фреймворк **Disco** [9]. Реализован он в сочетании Erlang и Python. Основные отличия его от Hadoop [10], связаны с тем, что Hadoop более строго следует исходной архитектуре MapReduce, описанной Google. Disco только основана на этой модели, но разработана так, чтобы быть легкой (за счет использования сильных сторон Erlang и Python). Disco предназначена для работы на кластере и обеспечивает как уровень отказоустойчивого планирования и выполнения, так и уровень распределенного и реплицированного хранилища.

Постановка задачи

Командой кафедры «Компьютерного моделирования и многопроцессорных систем» факультета «Прикладной математики и процессов управления» Санкт-Петербургского государственного университета был разработан новый C++ фреймворк Subordination [11]. Данный фреймворк даёт приемлемую универсальность с поддержкой полной отказоустойчивости, основываясь на модели управляющих объектов.

Учитывая этот факт и информацию выше, возникла необходимость в реализации интерфейса к новому фреймворку (далее SBN-Python), что позволило бы использовать все его преимущества на языке более высокого уровня Python.

Глава 1. Subordination: отказоустойчивость из «коробки»

1.1 Ключевые особенности

Отличительной особенностью нового фреймворка Subordination является предоставление максимально полной отказоустойчивости программам, написанным с использованием программного интерфейса системы: программы защищены от выхода из строя одного из подчиненных узлов, главного узла, а также всех узлов распределенной системы одновременно.

В новом интерфейсе программист не задумывается:

- по какому алгоритму параллельные части его программы распределяются по узлам распределенной системы, какие узлы при этом выбираются,
- на каком узле будет работать отдельно взятый фрагмент кода программы,
- как обработать выход из строя одного из узлов, и т.д.

1.2 Основные компоненты и принцип работы

Создание распределенной системы невозможно без создания головной программы, предоставляющей программный интерфейс. В нашем случае этой программой является сервис, запущенный на каждом узле распределенной системы, в задачи которого входят

- обнаружение новых узлов, а также узлов, вышедших из строя, и их включение или исключение из распределенной системы,
- выполнение задач и их пересылку на другие узлы системы.

Помимо этого, для эффективного распределения нагрузки сервис строит древовидную иерархию из сервисов (см. рис. 2), запущенных на других узлах системы. Каждый сервис определяет, сколько узлов находится за узлами, соединенными с ним непосредственно, и использует это значение для равномерного распределения управляющих объектов по всем узлам

системы, независимо от того, на каком узле была запущена программа изначально.

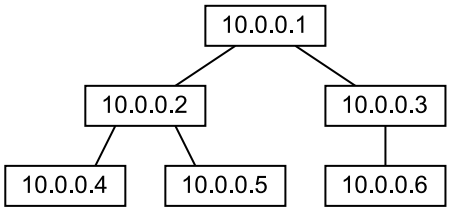


Рис. 2: Иерархия из сервисов.

Взаимодействие с описанным выше сервисом производится посредством программного интерфейса, написанного на низкоуровневом языке программирования, что позволяет максимально эффективно реализовать низкоуровневые абстракции для обеспечения отказоустойчивости и для использования в полной мере параллелизма, предоставляемого распределенной системой.

Низкоуровневый интерфейс основан на так называемых управляющих объектах (*kernel*) – объектах в языке программирования, которые содержат данные, которые необходимо обработать, и код для их обработки. За счёт этого управляющие объекты реализуют всю логику потока управления, накапливая в себе состояние текущей ветви. Конкретно, это происходит в их методах *act* и *react*. В методе *act* некоторая задача либо последовательно вычисляется, либо декомпозируется на подзадачи, представленные другим набором управляющих объектов (инициируется при помощи *upstream* метода). В методе *react* подчиненные объекты, завершившие работу, обрабатываются их родителем (инициируется при помощи *commit* метода) (см. рис. 3).

Для того чтобы содержащийся в объектах код был исполнен, они отправляются на конвейер – очередь из управляющих объектов, обработкой которой занимается пул потоков. В зависимости от типа конвейера поток, извлекая из него объект, может либо сразу начать выполнение содержащегося в нем кода локально, либо отправить объект на один из узлов кластера, где он опять попадет на конвейер и будет обработан локально. Все объекты помимо данных и кода содержат в себе ссылку на родительский объект.

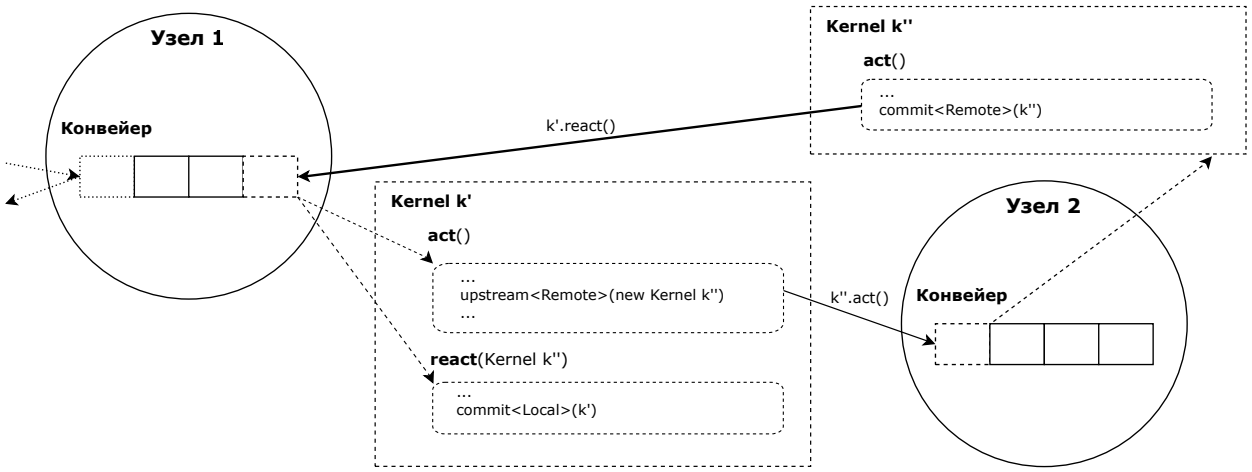


Рис. 3: Упрощенная схема выполнения.

Когда код дочернего объекта выполняется локально, система отправляет объект обратно к родительскому, даже если он находится на другом узле. Как только дочерний объект достигает родительского, родитель сохраняет результаты вычислений локально и дочерний объект удаляется. Описанная процедура взаимодействия объектов представляет собой аналог стека вызова функций, но для распределенных систем.

1.3 Возможность обработки сбоев

Связь родительский-дочерний объект используется для перезапуска дочерних объектов, которые были потеряны в результате выхода из строя узлов распределенной системы:

1. Когда сервис определяет, что ***один из примыкающих узлов вышел из строя***, все объекты, отправленные на этот узел, перенаправляются на один из оставшихся узлов (или выполняются локально, если таковых нет). Это возможно, поскольку сервис хранит в памяти все объекты, которые были отправлены на примыкающие узлы, которые еще не вернулись обратно.
2. Поскольку между объектами установлена строгая иерархическая связь, то возникает ***проблема перезапуска корня этой иерархии*** – объекта, у которого нет родительского. Для решения этой проблемы сервис автоматически копирует объект, являющийся корнем иерархии,

на один из примыкающих узлов. Если примыкающий узел выходит из строя, то программа продолжает работать в нормальном режиме, используя оригинал корневого объекта; если исходный узел выходит из строя, программа перезапускается, начиная с корневого объекта. Этот подход эффективен в предположении, что программа состоит из последовательных шагов, каждый из которых использует параллелизм внутри и глобальную синхронизацию всех параллельных потоков в конце шага: тогда перезапускается лишь один шаг вычислений, а не вся программа целиком.

3. Наконец, после ***выхода из строя всех узлов системы*** единственный шанс восстановить работу всех работавших поверх нее программ – считать их состояние из энергонезависимого хранилища (файловой системы). Для этого объекты, отправленные на примыкающие узлы, сохраняются не только в оперативной памяти узла, но и периодически записываются в файл. Когда система включается после выхода из строя всех узлов, этот файл считывается и все находящиеся в нем объекты восстанавливаются с учетом того, вернулись ли они на узел, или нет. Поскольку это происходит асинхронно на всех узлах системы, то гарантируется восстановление только корневых объектов, так как родительские объекты могут восстановиться позже дочерних.

   Подробнее о каждом сценарии можно узнать в статье [12].

Глава 2. SBN-Python: реализация и использование

2.1 Исполнение Python кода

Поскольку Python является своего рода интерфейсом, для описания, как и что он должен делать, существует соответствующая спецификация. На текущий момент есть довольно много реализаций этой спецификации [13]: cPython, IronPython, PyPy, Jython и многие другие.

Под «реализацией» Python следует понимать программу или среду, которая обеспечивает поддержку выполнения программ, написанных на языке Python.

Диаграмма на рис. 4 объясняет последовательность выполнения кода Python. Исходный код сначала компилируется и преобразуется в байт-код. Затем на виртуальной машине получившийся байт-код исполняется. Реализации Python определяются на основе языка, на котором построены эти виртуальные машины, или способа его интерпретации / компиляции.

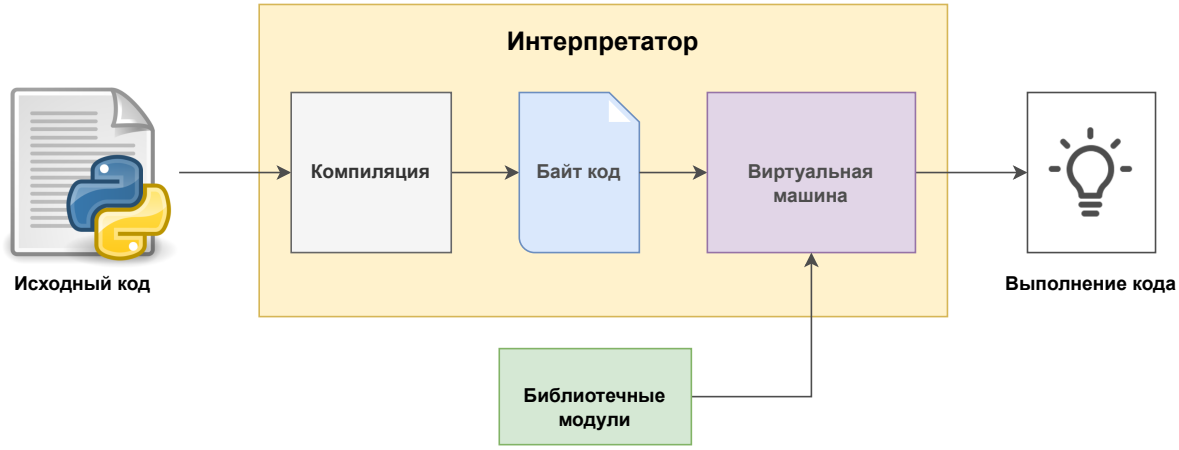


Рис. 4: Диаграмма исполнения кода Python.

Наиболее распространённой и де-факто эталонной реализацией Python на текущий момент является cPython [14]. В первую очередь, причина этого состоит в том, что байт-код, созданный компилятором этой реализации, запускается виртуальной машиной, написанной с использованием кода C. Поэтому степень совместимости с Python пакетами и модулями расширения C в cPython имеет высокое значение.

2.2 Python/C API и принцип использования

Для достижения поставленной задачи необходима низкоуровневая интеграция с исходным фреймворком, что может дать только разработка расширения интерпретатора Python.

В целях лучшей совместимости с C++ была использована cPython реализация. Она, в свою очередь, поставляется вместе с Python/C API [15] – инструментом, с помощью которого разработаны все примитивы и методы, необходимые для функционирования всего фреймворка.

Этот API дает программистам на C и C++ доступ к интерпретатору Python на различных уровнях. Есть две принципиально разные причины использования Python/C API:

1. Написание C++ модулей расширения интерпретатора Python.
2. Встраивание Python в C++ приложение.

В текущем случае необходимо использовать оба подхода: для работы с Python сценарием из C++ программы и, наоборот, использование C++ модуля в Python сценарии.

Под встраиванием Python понимается загрузка из C++ программы Python интерпретатора и передача ему Python сценария на исполнение. В данном случае C++ программа нужна, чтобы предварительно настроить все необходимые сервисы Subordination и сформировать первый управляющий объект.

При этом, как уже упоминалось ранее, программист, который разрабатывает Python сценарий, должен иметь возможность использовать C++ модуль для написания своих управляющих объектов и логики потока управления. Для этого используется уже принцип расширения. По итогу это выглядит, как привычное импортирование библиотеки и использования её структур данных и методов.

2.3 Получившийся интерфейс

Получившийся интерфейс, как и исходная система Subordination, основан на так называемых управляющих объектах.

Запуск Python программы осуществляется через расширенный интерпретатор (далее *sbn-python3*). Он в свою очередь поставляет из «коробки» C++ библиотеку *sbn*, которая включает следующие компоненты:

- Класс *sbn.Kernel*, позволяющий определять свои управляющие объекты, используя принцип наследования.
- Управляющие функции, такие как *sbn.upstream* и *sbn.commit*, маркер *sbn.Target*, задающий локальный или удалённый узел, – всё то, что позволяет определять логику потока управления.

В наследнике класса *sbn.Kernel* имеется возможность определить функции *act* и *react*, а также любые поля, что позволяет нам задавать данные, которые необходимо обработать, а также код для их обработки.

Пример *hello-world программы* (далее *hello.py*) представлен на листинге 1.

```
import sbn

class Child(sbn.Kernel):
    def __init__(self, n: int):
        super(Child, self).__init__()
        self.n = n
    def act()(self):
        self.answer = "Hello! I am the %i child" % self.n
        sbn.commit(kernel=self, target=sbn.Target.Remote)

class Main(sbn.Kernel):
    def __init__(self, *args, **kwargs):
        super(Main, self).__init__(*args, **kwargs)
        self.number_child = 3
        self.counter = 0
    def act()(self):
        for i in range(1, self.number_child + 1):
            sbn.upstream(parent=self, child=Child(i), target=sbn.Target.
   Remote)
    def react()(self, k: Child):
        print("Hello, who are you?", k.answer)
        self.counter += 1
        if self.counter == self.number_child:
            sbn.commit(kernel=self, target=sbn.Target.Local)
```

Листинг 1: Пример Hello-world программы

Теперь верхнеуровнево пройдёмся по всем шагам исполнения *hello.py* программы через *sbn-python3* (схематичный вид см. в Приложении А):

Узел 1

1. Запуск *sbn-python3* с передачей пути к Python сценарию *hello.py*

   (a) Загружается Python интерпретатор;
   (b) Регистрируются два типа управляющего C++ объекта: *Kernel* (далее ***C++ Kernel***) и *Main* (наследник Kernel, но с переопределённым методом act; далее ***C++ Main***);
   (c) Создаётся первый управляющий объект *C++ Main* со всеми входными аргументами, и его вызов метода ***act*** отправляется на исполнение на конвейер.

2. Исполнение *C++ Main.act()*:

   (a) Пул потоков забирает *C++ Main.act()* с конвейера и исполняет его;
   (b) В методе *act* у *C++ Main* из Python сценария загружается определение ***Python Main*** (наследник *sbn.Kernel*) и создаётся Python объект;
   (c) Далее C++ и Python объект обмениваются ссылками друг на друга;
   (d) У *Python Main* вызывается метод ***act***.

3. Исполнение *Python Main.act()* (т.е только с этого шага начинает исполняться Python сценарий):

   (a) В методе *act* у *Python Main* создаются дочерние управляющие объекты ***Python Child*** (наследники *sbn.Kernel*);
   (b) В конструкторе *sbn.Kernel* создаётся экземпляр ***C++ Kernel***, с которым Python объект обменивается ссылками;
   (c) В том же методе *act* у *Python Main* созданные объекты передаются в функцию ***sbn.upstream*** для передачи на конвейер на исполнение;

(d) В функции *sbn.upstream* из *Python Child* достаётся истинный управляющий объект *C++ Kernel* и *upstream* вызывается уже для него.

4. Что происходит после вызова *C++ upstream* (для дочернего объекта):

   (a) Как видно по коду, эти объекты должны быть перенесены на удалённый узел посредством маркера ***sbn.Target.Remote***;
   (b) Для передачи на другой узел, у *C++ Kernel* сначала вызывается метод ***write***, в котором всё содержимое *sbn.Kernel* записывается в бинарное представление;
   (c) Далее на удалённом узле у того же *C++ Kernel* вызывается метод ***read***, куда приходит бинарное представление *sbn.Kernel* и там же восстанавливается;
   (d) После чего эти *C++ Kernel* объекты кладутся на исполнение на конвейер (вызов его метода *act*) уже на 2 узле.

Узел 2

5. Исполнение *C++ Kernel.act()* (т.е. дочернего объекта):

   (a) Пул потоков снимает с конвейера *C++ Kernel.act()* и вызывает его;
   (b) В этом методе вызывается метод *act* у *Python Child*, который может получен по ссылке.

6. Исполнение *Python Child.act()*:

   (a) Выполняется какая-то бизнес логика (в данном случае это формирование hello-ответа);
   (b) В конце метода *act* у *Python Child* объекта вызывается метод ***sbn.commit*** для возвращения объекта родителю;
   (c) В методе *sbn.commit* у *sbn.Kernel* достаётся ссылка на C++ Kernel и уже конкретно для него вызывается *commit*.

7. Что происходит после *C++ commit* для *C++ kernel*:

(a) Та же процедура *write/read*;

(b) На 1 узле на конвейер кладётся на исполнение метод ***react*** у *C++ Main*.

Узел 1

8. Исполнение *C++ Main.react()*:

   (a) Пул потоков снимает метод *react* объекта *C++ Main* и вызывает его с передачей дочернего объекта *C++ Kernel*;

   (b) По ссылке достаётся *Python Child*;

   (c) Далее в этом методе, так же как и в act, вызывается метод *react* у *Python Main*, куда пересылаются дочерние объекты в виде *sbn.Kernel* (*Python Child*).

9. Исполнение *Pyhton Main.react()*:

   (a) После того, как все дочерние объекты вернулись, *Python Main* сигнализирует о завершении, вызовом *commit*.

2.4 Обсуждение

Для переноса управляющего объекта между узлами, всё его состояние, которое он в себе хранит, должно быть сначала упаковано в бинарный вид, а потом восстановлено уже на новом узле. За счёт того, что в Python присутствует полная интроспекция (возможность получения всей информации о внутренней структуре любого объекта) это процесс можно унифицировать. В текущем случае были задействованы средства Pickle [16] пакета, который разбирает объект на свои составляющие и преобразует их в бинарный вид. Тоже самое повторяется и в обратную сторону. В исходном C++ фреймворке это приходилось прописывать явно и для каждого управляющего объекта по отдельности.

В текущей реализации основной сервис всё также взаимодействует именно с управляющими объектами в C++, которые в свою очередь уже, используя интерпретатор, работают с Python объектами, вызывая их методы. Тут стоит упомянуть проблему, с которой пришлось столкнуться:

GIL (Global Interpreter Lock) [17] – это своеобразная блокировка, которая присутствует в cPython реализации, за счёт которой только одному потоку позволено управлять интерпретатором Python.

Для корректного управления памятью в Python существует некоторая альтернатива сборщику мусора. Иными словами, для каждого созданного объекта в Python заводится переменная с целью подсчёта всех ссылок на данный объект. При обнулении этой переменной, выделенная ранее память под этот объект освобождается. В случае многопоточного приложения возникает проблема, что сразу несколько потоков могут увеличивать или уменьшать значение этой переменной, что потенциально может привести к тому, что память будет очищена неправильно и удалится тот объект, на который ещё существует ссылка. Эту проблему и был призван решить GIL.

На текущий момент было решено организовать работу на одном узле в один процесс. В будущем есть планы попробовать воспользоваться довольно популярным решением, а именно под каждый процесс заводить свой Python интерпретатор с отдельно выделенной памятью (как это сделано в Python модуле multiprocessing [18]), что позволит обойти проблему с GIL. Но это потребует много усилий на реализацию согласованности состояний интерпретаторов между процессами.

Есть ещё более кардинальное решение – смена интерпретатора. Но такой вариант тут плохо подходит, т.к. в текущей реализации используется Python/C API, которая в свою очередь не совместима с иными вариантами Python интерпретаторов.

Глава 3. SBN-Python: тестирование

3.1 Программа

С целью замера производительности и проверки всех сценариев отказоустойчивости была реализована задача пакетной обработки по восстановлению частотно-направленного спектра морского волнения и вычислению дисперсии (область под графиком спектра).

The National Data Buoy Center (NDBC) поддерживает сеть данных метеобуев для мониторинга океанографических и метеорологических данных у берегов США. Волновые данные NDBC представляют большой интерес из-за необходимости учёта волн при морских операциях, а также часто используются для различных инженерных и научных применений.

Были взяты исторические данные, которые представляют собой записи частотно-направленного спектра морского волнения, записанные с помощью метеобуев и закодированных в пяти переменных. Каждая запись содержит дату измерения и дискретные значения спектра для каждой из частот (диапазон частот и их количество фиксировано и одинаково во всех файлах). Каждый файл содержит записи только одного из метеобуя за промежуток времени равный одному году (см. рис. 5).

Рис. 5: Структура данных задачи.

Программа сканирует входной каталог на наличие файлов и группирует их по году и номеру метеобуя. Поскольку спектр закодирован с

помощью пяти переменных и данные по каждой из переменных находятся в отдельных файлах, то перед обработкой данные из каждой пятерки файлов с помощью отдельного управляющего объекта объединяются в один кортеж. Затем в дочерних объектах записи внутри каждой пятерки соединяются по дате измерения, а рассчитывается дисперсия при помощи формулы 1 [19]:

$$S(\omega,\theta) = \frac{1}{\pi}\left[\frac{1}{2} + r_1\cos(\theta-\alpha_1) + r_2\sin(2(\theta-\alpha_2))\right] S_0(\omega), \qquad (1)$$

где ω и θ есть частота и направление волны, а $r_{1,2}$, $\alpha_{1,2}$ и S_0 - параметры, взятые из измерений. После вычисления всех дисперсий программа завершается.

В выборке может не оказаться данных для какой-то из переменных или для определенных дат. Неполные данные отсеиваются, поскольку по ним невозможно восстановить спектр. Также, данные для некоторых дат могут дублироваться, эти случаи также обрабатываются.

Итоговый алгоритм представлен на рис. 6

Рис. 6: Алгоритм программы.

3.2 Замеры производительности

Сначала проводилось тестирование на производительность без сбоев, как для C++, так и для Python реализации. Количество задействованных узлов в кластере изменялось с 1 до 6. Тест повторялся три раза, затем полученные результаты усреднялись. На вход подавались данные из NDBC [20] за год и за 10 лет (см. таблицу 1).

Таблица 1: Свойства набора данных NDBC

Свойства	**За год**	**За 10 лет**
Размер данных	43МБ	512МБ
Размер распакованных данных	238МБ	2851МБ
Количество станций	25	25
Общее количество спектров	118 491	1 637 809

Поскольку не стояло целью измерение накладных расходов параллельной файловой системы, те же входные данные были скопированы в локальную файловую систему каждого узла кластера.

Результат представлен на рис. 7 и в таблице 2.

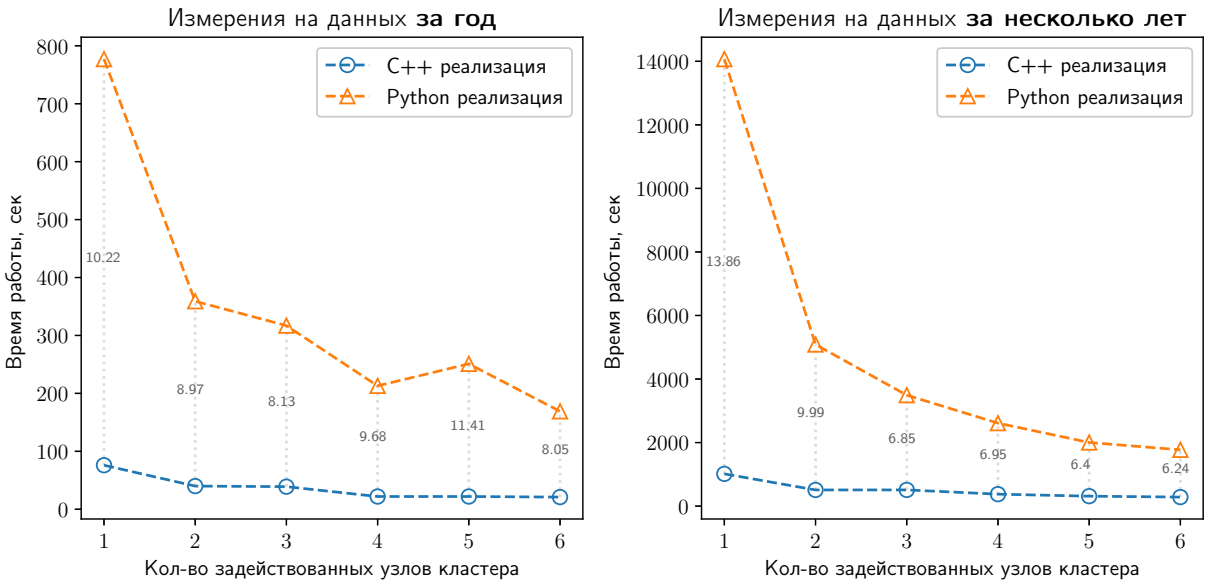


Рис. 7: Замер производительности C++ и Python реализации без сбоев.

Таблица 2: Замер производительности C++ и Python реализации без сбоев (в сек).

Данные	Реализация	Кол-во узлов					
		1	2	3	4	5	6
За год	C++	76	40	39	22	22	21
	Python	777	359	317	213	251	169
За 10 лет	C++	1015	509	510	376	313	284
	Python	14069	5086	3492	2612	2003	1772

Далее с целью повторить сценарии отказоустойчивости для Python реализации на данных за год таким же образом были произведены замеры производительности с симулированием сбоя узла, на котором находится главный управляющий объект (*superior-сбой*), его копия (*copy-superior-сбой*) или подчинённый объект (*subordinate-сбой*), а также выход из строя всех узлов (*all-сбой*) в сравнении с выполнением программы без сбоев (см. рис. 8 и таблицу 3).

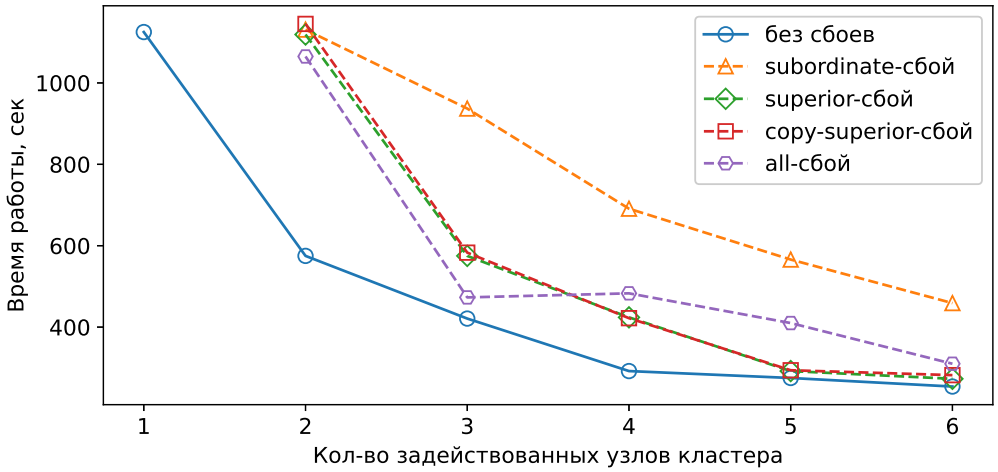


Рис. 8: Замер производительности Python реализации со сбоями.

Таблица 3: Замер производительности Python реализации со сбоями (в сек).

Запуск	Кол-во узлов					
	1	2	3	4	5	6
без сбоев	1125	575	421	292	275	254
subordinate-сбой		1131	937	691	566	459
superior-сбой		1119	575	424	292	273
copy-superior-сбой		1145	583	422	294	282
all-сбой		1065	473	483	410	310

3.3 Обсуждение

Замедление Python реализации было ожидаемым. Во-первых, это связано с с обсуждаемым ранее GIL, который позволяет ему работать только в однопоточном режиме. Во-вторых, для сериализации/десериализации управляющих объектов используется не быстрые операции интроспекции. Кроме того, сами Python вычисления не так оптимизированы, как их аналоги в C++. С ростом количества данных и задействованных узлов коэффициент замедления (отношение времени Python реализации к времени C++ реализации) в среднем уменьшается с 12 до 6.

Случай со сбоями узлов с главным управляющим объектом или его копией показал, что это приводит к снижению производительности до производительности кластера без отказавшего узла. Сбой узлов с подчинёнными объектами даёт тот же эффект, но с большими накладными расходами из-за их количества. При выходе из строя всех узлов добавляется некоторое время восстановления.

Глава 4. SBN-Python: применение на реальном кейсе компании

4.1 Задача

Компания ООО «Газпромнефть-ЦР» является одним из драйверов Цифровой трансформации «Газпром нефти» и сейчас активно занимается задачами по обработке и анализу данных. В связи с чем, проблема наличия универсального отказоустойчивого интерфейса на Python для распределенных вычислений для неё также актуальна. Внедрение новых решений может помочь начать решать задачи бизнеса лучше и эффективнее.

Поэтому в рамках научно-исследовательской практики была проверена возможность применения нового интерфейса на реальном кейсе. Конкретно рассматривалась задача шумоподавления в данных сейсморазведки [21], которая является обязательной, но самой длительной операцией в процессе обработки данных.

Иначе говоря, стояла задача разобрать текущее решение, продумать и реализовать новую архитектуру с использованием SBN-Python, и в конечном счёте развёрнуть получившиеся решение на мощностях компании с целью проверки работоспособности и замера производительности.

4.2 Разбор текущего решения

В рамках проекта НИОКР командой компании были разработаны программные модули и выстроен рабочий процесс, позволяющий производить необходимые трасформации над сейсмическим кубом для подавления имеющихся в нём шумов, возникших при сейсмосъёмке.

В результате разбора исходного кода текущего решения была выявлена следующая схема его работы (см. рис. 9):

1. Загрузка SGY (Society of Exploration Geophysicists) файла [22] с данными сейсмического куба в двоичном и текстовом формате, включающих в себя координаты отраженных сейсмических волн;
2. Параллельная декомпозиция с помощью multiprocessing [18] SGY файла на NPY файлы [23], каждый из которых содержит NumPy данные

по различным источникам взрыва, и сохранение в файловую систему;

3. Чтение NPY файлов, разбиение на блоки заданного размера с последующей параллельной обработкой на наборе трасформаторов с помощью multiprocessing и сохранение результатов в файловую систему;

4. Чтение обработанных NPY файлов и последовательное составление итогового SGY файла.

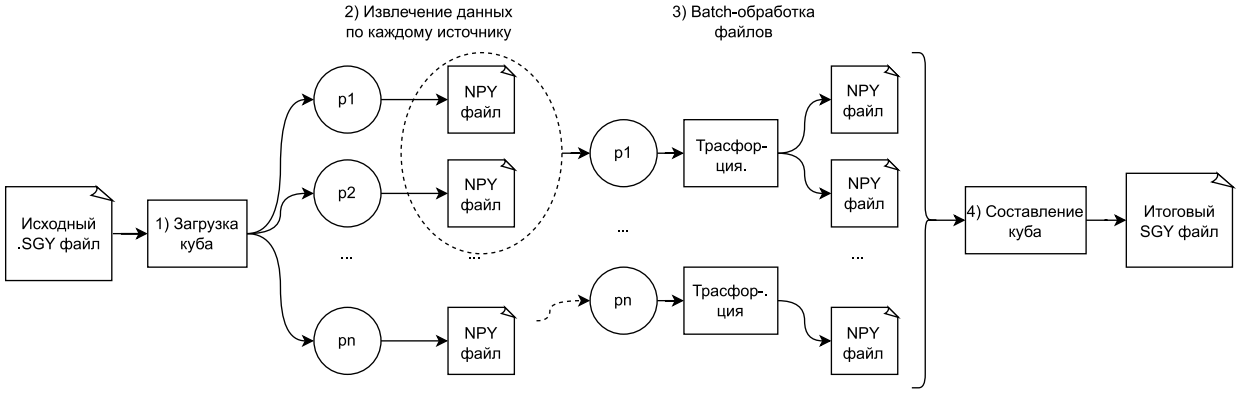


Рис. 9: Схема работы текущего решения.

Из-за того, что данные задачи достаточно большого объёма, весь процесс проходит достаточно долго. Применение multiprocessing-а хоть и даёт ускорение, но не гарантирует отказоустойчивость. В следствии чего в текущем решении имеются следующие особенности, что для данной задачи неэффективно:

1. Интенсивная работа с файловой системой для сохранения промежуточных результатов;
2. Синхронизация процессов на каждом шаге (модель Bulk Synchronous Parallel [24]).

4.3 Построение новой архитектуры решения

Исходя из изложенного ранее принципа использования нового интерфейса была построена новая архитектура решения (см. рис. 10):

1. Программа начинает свою работу с *Main Kernel*. В методе **act** инициируется *SgyProcess Kernel* и ему передаётся путь к SGY файлу;

2. В методе **act** *SgyProcess Kernel*-а SGY файл загружается и инициируются *BatchProcess Kernel*-ы с номерами источников взрыва;
3. В методе **act** каждого *BatchProcess Kernel*-а извлекаются данные по соответствующим источникам и далее обрабатываются набором трасформеров;
4. В метод **react** *SgyProcess Kernel*-а приходит результат работы *BatchProcess Kernel*-а и сохраняется в итоговый SGY файл;
5. В методе **react** *Main Kernel*-а программа завершается.

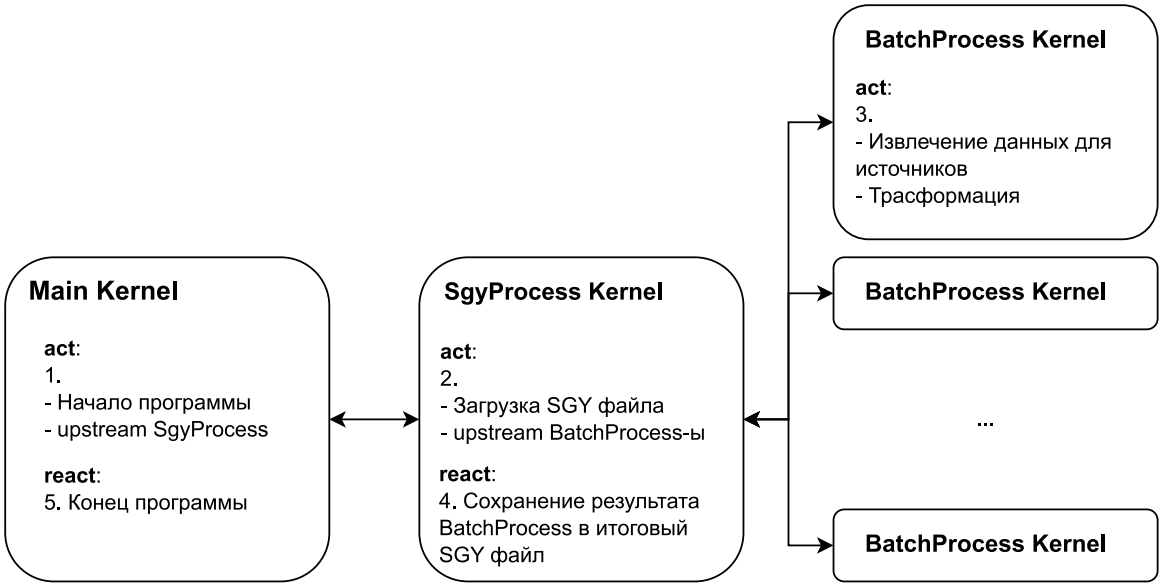


Рис. 10: Новая архитектура решения.

Тем самым новая архитектура позволяет произвести отказоустойчивую распределенную обработку сейсмического куба в оперативной памяти [25] и в модели реактивного программирования [26].

4.4 Реализация и развёртка получившегося решения

Расширение интерпретатора языка Python идёт вместе с библиотекой sbn, в которой содержатся все необходимые примитивы и методы для

реализации решения в представленной ранее архитектуре. В качестве иллюстрации на листинге 2 приведён пример реализации *SgyProcess Kernel*-а.

```
import sbn

class SgyProcess(sbn.Kernel):
    def __init__(self, input_path_sgy:str=None, output_path_sgy:str=None,
   batch_size:int=None):
        super(SgyProcess, self).__init__()
        self.input_path_sgy = input_path_sgy
        self.output_path_sgy = output_path_sgy
        self.batch_size = batch_size
        self._count_ready = 0

    def act(self):
        # Read input .sgy file
        self.in_segy = self._read_input_sgy()
        self.sources = self.segy.get_sources()
        self._count = len(self.sources)

        # Create pipeline
        pipeline = self._create_transformer_pipeline()

        # Init batch processing
        for i in range(0, self._count, self.batch_size):
            sbn.upstream(self,
                BatchProcess(
                    self.in_segy,
                    self.sources[i:i + self.batch_size], pipeline
                ),
                target=sbn.Target.Remote)

    def react(self, child: BatchProcess):
        for result in child.batch_results:
            self._save_to_sgy(result.data, result.id)
        self._count_ready += self.batch_size
        if self._count_ready == self._count:
            sbn.commit(self, target=sbn.Target.Remote)
```

Листинг 2: Реализация *SgyProcess Kernel*-а

Для работы всего фреймворка на мощностях компании потребовалось собрать все необходимые компоненты в бинарный вид и упаковать в установочный пакет при помощи Guix [27].

4.5 Замер производительности

Результаты замера производительности показали схожую динамику, что и в задаче по восстановлению спектра морского волнения (см. рис. 11 и таблицу 4).

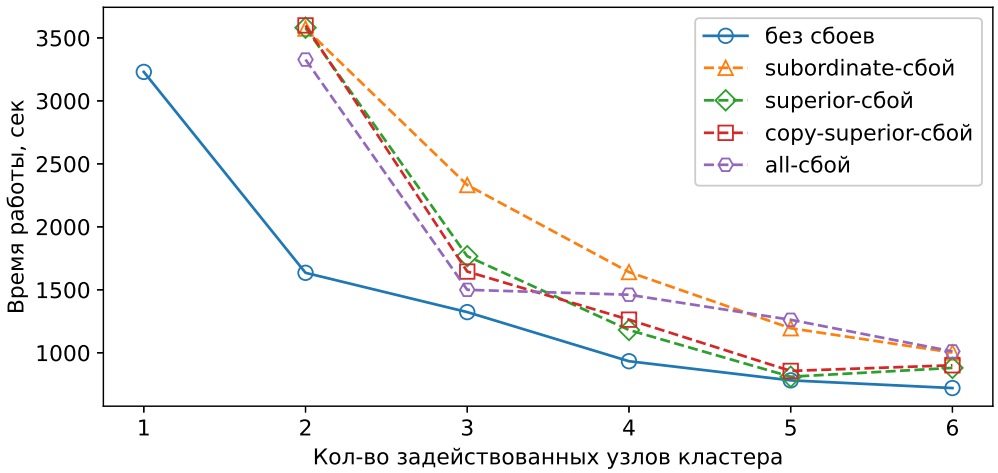


Рис. 11: Замер производительности решения с SBN-Python.

Таблица 4: Замер производительности решения с SBN-Python (в сек).

Запуск	**Кол-во узлов**					
	1	2	3	4	5	6
без сбоев	3230	1635	1323	933	781	720
subordinate-сбой		3574	2332	1641	1195	1002
superior-сбой		3581	1767	1181	810	880
copy-superior-сбой		3599	1645	1263	856	901
all-сбой		3328	1500	1461	1262	1011

Заключение

В данной дипломной работе мной был разработан высокоуровневый интерфейс на языке Python для нового фреймворка отказоустойчивых распределённых вычислений Subordination. Получившиеся решение, как и планировалось, соответствует тем же принципам построения распределенного приложения, что и в исходной системе.

Результаты проведения тестирования показали корректность работы интерфейса, прирост производительности при увеличении количества узлов в кластере, а также возможность обработки различных сценариев сбоя узлов за приемлемое время.

Помимо этого, новый интерфейс для отказоустойчивых распределенных вычислений SBN-Python, во-первых, был успешно применён на реальном кейсе компании ООО «Газпромнефть-ЦР», во-вторых, смог показать свои преимущества по сравнению с текущим решением.

В дальнейшем планируется, во-первых, лучше проработать текущее решение, во-вторых, продумать и реализовать возможность использования полученного интерфейса для построения распределенных веб-сервисов.

Список литературы

[1] Parallel Processing and Multiprocessing in Python [Электронный ресурс] URL: https://wiki.python.org/moin/ParallelProcessing (дата обращения: 20.11.2020)

[2] A Gentle Introduction to Ray [Электронный ресурс] URL: https://docs.ray.io/en/master/ray-overview/index.html (дата обращения: 25.01.2021)

[3] Moritz P., Nishihara R., Wang S., Tumanov A., Liaw R., Liang E., Paul W., Michael I., Stoica I. Ray: A Distributed Framework for Emerging AI Applications // In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18). 2018.

[4] John C., Mitchell. Concepts in programming languages // Cambridge University Press. 2003. P. 529.

[5] Ray 1.0 Architecture [Электронный ресурс] URL: https://docs.google.com/document/d/1lAy0Owi-vPz2jEqBSaHNQcy2IBSDEHyXNOQZlGuj93c/preview# (дата обращения: 27.11.2020)

[6] Babuji Y., Chard K., Foster I., Katz D. S., Wilde M., Woodard A., Wozniak J. Parsl: Scalable Parallel Scripting in Python // In 10th International Workshop on Science Gateways (IWSG '18). CEUR-WS.org. 2018.

[7] Babuji Y., Woodard A., Li Z., Katz D. S., Clifford B., Kumar R., Lacinski L., Chard R., Wozniak J., Foster I., Wilde M., Chard K. Parsl: Pervasive Parallel Programming in Python // In 28th ACM International Symposium on HighPerformance Parallel and Distributed Computing (HPDC). 2019.

[8] Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters // In 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04). 2004. P. 137-150.

[9] Mundkur P., Tuulos V., Flatow J. Disco: a computing platform for large-scale data analytics // Erlang '11: Proceedings of the 10th ACM SIGPLAN workshop on ErlangSeptember. 2011. P. 84-89.

[10] Гусейнов А., Бочкова И. Исследование распределенной обработки данных на примере системы Hadoop. 2016.

[11] Gankevich I., Tipikin Y., Gaiduchok V. Subordination: Cluster management without distributed consensus // In International Conference on High Performance Computing Simulation (HPCS). 2015. P. 639-642.

[12] Gankevich I., Tipikin Y., Korkhov V. Subordination: Providing resilience to simultaneous failure of multiple cluster nodes // In Proceedings of International Conference on High Performance Computing Simulation (HPCS'17). 2017. P. 832–838.

[13] Python Implementations [Электронный ресурс] // URL: https://wiki.python.org/moin/PythonImplementations (дата обращения 01.10.2020)

[14] cPython [Электронный ресурс] URL: https://github.com/python/cpython (дата обращения 17.10.2020)

[15] Python/C API Reference Manual [Электронный ресурс] URL: https://docs.python.org/3/c-api/index.html (дата обращения: 05.10.2020)

[16] Pickle – Python object serialization [Электронный ресурс] URL: https://docs.python.org/3/library/pickle.html (дата обращения: 11.10.2020)

[17] Python Global Interpreter Lock [Электронный ресурс] URL: https://tproger.ru/translations/global-interpreter-lock-guide/ (дата обращения: 15.10.2021)

[18] Multiprocessing – process-based parallelism [Электронный ресурс] // URL: https://docs.python.org/3/library/multiprocessing.html (дата обращения 21.10.2020)

[19] Earle M. D. Nondirectional and Directional Wave Data Analysis Procedures // Louisiana: Neptune Sciences, Inc. 1996. P. 37.

[20] National Data Buoy Center [Электронный ресурс] URL: http://www.ndbc.noaa.gov/dwa.shtml (дата обращения: 07.12.2020)

[21] Геофизические методы исследований // Сейсморазведка [Электронный ресурс] URL: https://www.geokniga.org/sites/geokniga/files/inbox/1209/chapter1.pdf (дата обращения: 10.03.2021)

[22] SEG-Y [Электронный ресурс] URL: https://ru.qaz.wiki/wiki/SEG-Y (дата обращения: 20.03.2021)

[23] NPY format [Электронный ресурс] URL: https://numpy.org/devdocs/reference/generated/numpy.lib.format.html (дата обращения: 01.04.2021)

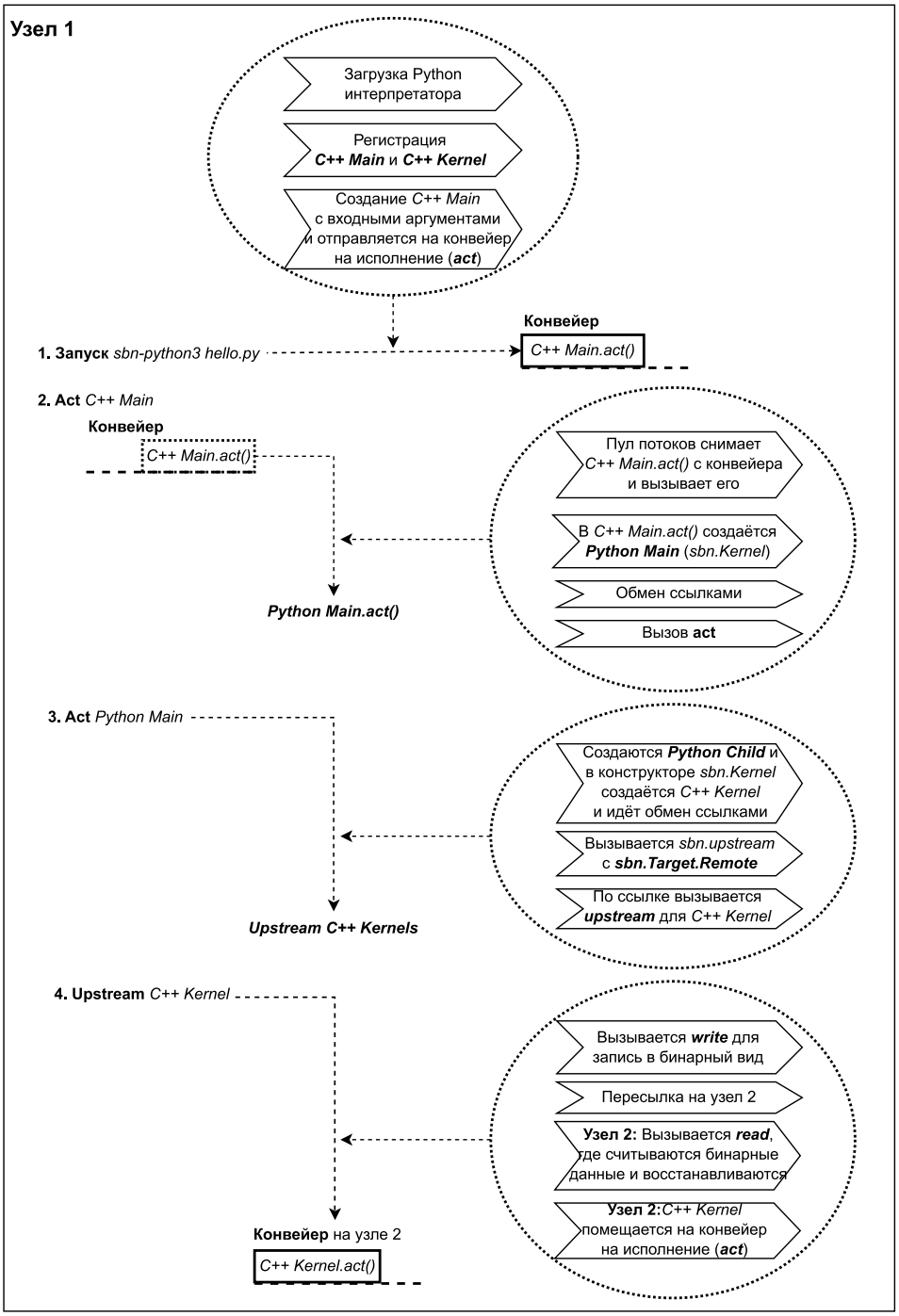
[24] Bulk Synchronous Parallel [Электронный ресурс] URL: https://ieeexplore.ieee.org/document/552669 (дата обращения: 10.04.2021)

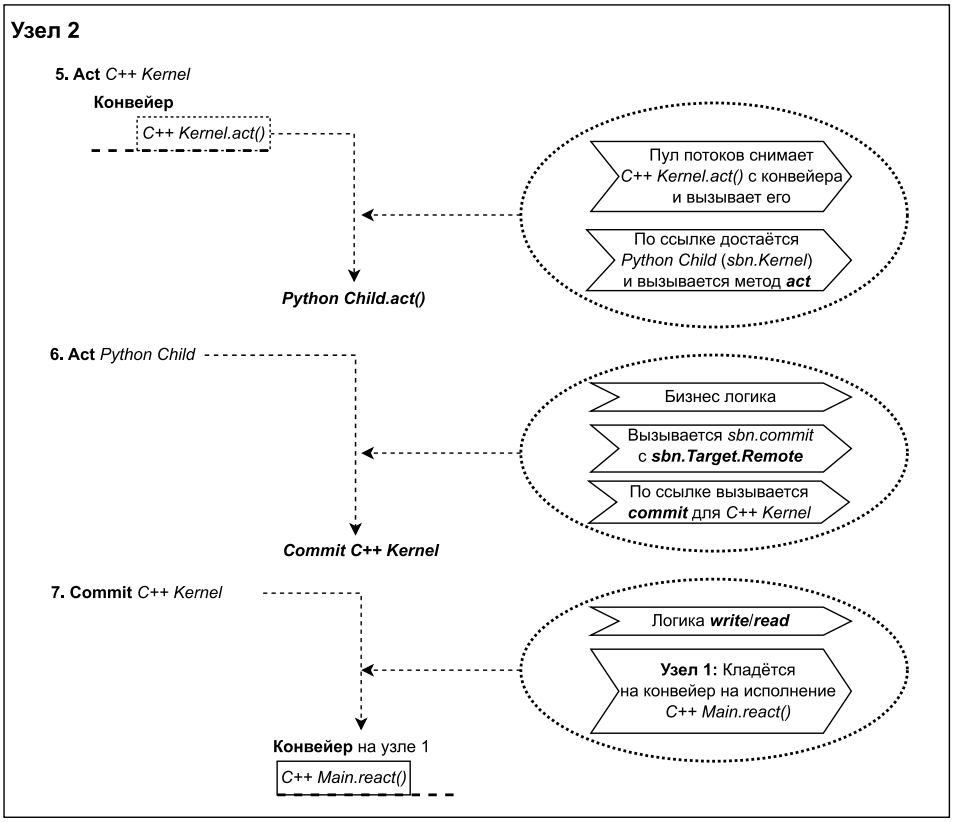
[25] Технология In-Memory Computing [Электронный ресурс] URL: https://www.crn.ru/news/detail.php?ID=124497 (дата обращения: 13.04.2021)

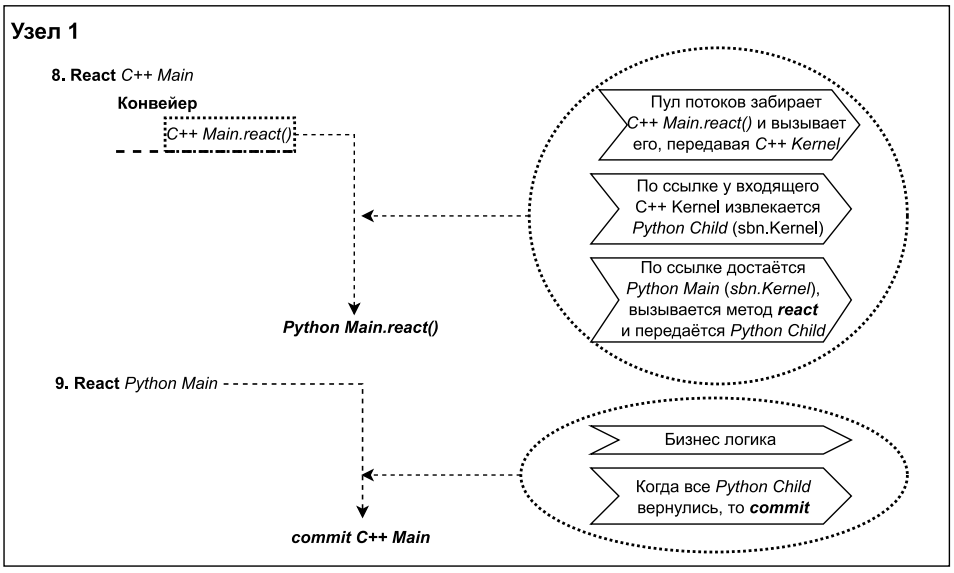
[26] Введение в реактивное программирование [Электронный ресурс] URL: https://habr.com/ru/company/arcadia/blog/432004/ (дата обращения: 17.04.2021)

[27] Invoking guix pack [Электронный ресурс] URL: https://guix.gnu.org/manual/en/html_node/Invoking-guix-pack.html (дата обращения: 20.04.2021)

Приложение А. Схема работы SBN-Python Hello программы




Узел 2
5. Act C++ Kernel
Конвейер
C++ Kernel.act()
Пул потоков снимает C++ Kernel.act() с конвейера и вызывает его
По ссылке достаётся Python Child (sbn.Kernel) и вызывается метод act
Python Child.act()
6. Act Python Child
Бизнес логика
Вызывается sbn.commit с sbn.Target.Remote
По ссылке вызывается commit для C++ Kernel
Commit C++ Kernel
7. Commit C++ Kernel
Логика write/read
Узел 1: Кладётся на конвейер на исполнение C++ Main.react()
Конвейер на узле 1
C++ Main.react()


Узел 1
8. React C++ Main
Конвейер
C++ Main.react()
Пул потоков забирает C++ Main.react() и вызывает его, передавая C++ Kernel
По ссылке у входящего C++ Kernel извлекается Python Child (sbn.Kernel)
По ссылке достаётся Python Main (sbn.Kernel), вызывается метод react и передаётся Python Child
Python Main.react()
9. React Python Main
Бизнес логика
Когда все Python Child вернулись, то commit
commit C++ Main