DOI: 10.15514/ISPRAS–2021–33(1)–5

Цифровые двойники в туманных вычислениях: организация обработки данных с сохранением состояния на базе микропотоков работ

[1] *А.Б.А. Алаасам, ORCID: 0000-0002-2084-8899 <alaasamab@susu.ru>*
[1] *Г.И. Радченко, ORCID: 0000-0002-7145-5630 <gleb.radchenko@susu.ru>*
[1,2,3] *А.Н. Черных, ORCID: 0000-0001-5029-5212 <chernykh@cicese.mx>*
[4] *Х.Л. Гонсалес-Компеан, ORCID: 0000-0002-2160-4407 <jgonzalez@tamps.cinvestav.mx>*

[1] *Южно-Уральский государственный Университет,*
454080, Россия, г. Челябинск, проспект Ленина, д. 76.
[2] *Центр научных исследований и высшего образования*
Мексика, 22860, Нижняя Калифорния, Энсенада, ш. Тихуана-Энсенада, 3918
[3] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, г. Москва, ул. А. Солженицына, дом 25
[4] *Центр перспективных исследований Национального политехнического института,*
Мексика, 87130, Сьюдад-Виктория, Тамаулипас

Аннотация. Цифровые двойники процессов и устройств используют информацию, получаемую с датчиков, для синхронизации своего состояния с сущностями физического мира. Концепция потоковых вычислений позволяет эффективно обрабатывать события, генерируемые такими датчиками. Однако необходимость отслеживания состояния экземпляра объекта приводит к невозможности организации цифровых двойников в виде сервисов без сохранения состояния. Еще одной особенностью цифровых двойников является то, что некоторые задачи, реализованные на их основе, требуют способности реагировать на входящие события на скорости, близкой к реальному времени. В этом случае использование облачных вычислений становится неприемлемым из-за высокой временной задержки. Туманные вычисления решают эту проблему, перемещая некоторые вычислительные задачи ближе к источникам данных. Однако сложности в организации обработки состояния на базе контейнеризованных мироксервисных систем создает проблемы в обеспечении бесперебойной работы таких сервисов в условиях туманных вычислений. Таким образом, основная задача исследования заключается в создании методов контейнеризированной обработки потоков данных с сохранением состояния на основе микросервисов, для поддержки развертывания компонентов цифровых двойников на базе туманных вычислительных сред. В рамках этой статьи мы исследуем возможности живой миграции процессов обработки потоков данных с сохранением состояния и способы перераспределения вычислительной нагрузки между облачными и туманными узлами с использованием платформы Kafka и Kafka Stream DSL API.

Ключевые слова: цифровые двойники; микросервисы; микропотоки работ; потоковая обработка данных; контейнеры; Apache Kafka; облачные вычисления; туманные вычисления

Для цитирования: Алаасам А.Б.А., Радченко Г.И., Черных А.Н., Гонсалес-Компеан Х.Л. Цифровые двойники в туманных вычислениях: организация обработки данных с сохранением состояния на базе микропотоков работ. Труды ИСП РАН, том 33, вып. 1, 2021 г., стр. 65-80. DOI: 10.15514/ISPRAS–2021–33(1)–5

Благодарности: исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-37-90073.

Stateful Stream Processing Containerized as Microservice to Support Digital Twins in Fog Computing

[1] *A.B.A. Alaasam, ORCID: 0000-0002-2084-8899 <alaasamab@susu.ru>*
[1] *G. Radchenko, ORCID: 0000-0002-7145-5630 <gleb.radchenko@susu.ru >*
[1,2,3] *A. Tchernykh, ORCID: 0000-0001-5029-5212 <chernykh@cicese.mx>*
[4] *J.L. Gonzalez-Compean, ORCID: 0000-0002-2160-4407 <jgonzalez@tamps.cinvestav.mx>*

[1] *South Ural State University,*
454080, Russia, Chelyabinsk, Lenin Avenue, 76
[2] *Centro de Investigación Científica y de Educación Superior,*
3918, Ensenada-Tijuana Highway, Ensenada, 22860, Mexico
[3] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[4] *Center for Research and Advanced Studies of the National Polytechnic Institute,*
Tamaulipas, Ciudad Victoria 87130, Mexico

Abstract. Digital twins of processes and devices use information from sensors to synchronize their state with the entities of the physical world. The concept of stream computing enables effective processing of events generated by such sensors. However, the need to track the state of an instance of the object leads to the impossibility of organizing instances of digital twins as stateless services. Another feature of digital twins is that several tasks implemented on their basis require the ability to respond to incoming events at near-real-time speed. In this case, the use of cloud computing becomes unacceptable due to high latency. Fog computing manages this problem by moving some computational tasks closer to the data sources. One of the recent solutions providing the development of loosely coupled distributed systems is a Microservice approach, which implies the organization of the distributed system as a set of coherent and independent services interacting with each other using messages. The microservice is most often isolated by utilizing containers to overcome the high overheads of using virtual machines. The main problem is that microservices and containers together are stateless by nature. The container technology still does not fully support live container migration between physical hosts without data loss. It causes challenges in ensuring the uninterrupted operation of services in fog computing environments. Thus, an essential challenge is to create a containerized stateful stream processing based microservice to support digital twins in the fog computing environment. Within the scope of this article, we study live stateful stream processing migration and how to redistribute computational activity across cloud and fog nodes using Kafka middleware and its Stream DSL API.

Keywords: digital twins; microservices; micro-workflows; stream processing; containers; Apache Kafka; cloud computing; fog computing

For citation: Alaasam A.B.A., Radchenko G., Tchernykh A., Gonzalez-Compean J.L. Stateful Stream Processing Containerized as Microservice to Support Digital Twins in Fog Computing. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 1, 2021, pp. 65-80 (in Russian). DOI: 10.15514/ISPRAS–2021–33(1)–5

Acknowledgements. The reported study was funded by RFBR, project number 19-37-90073.

1. Введение

Технологии интернета вещей (Internet of Things, IoT) обеспечивают интеграцию объектов повседневного мира в глобальную вычислительную среду [1]. Из-за широкого распространения технологий IoT физический и цифровой миры становятся все более взаимосвязанными. Всё чаще возникают задачи организации двусторонней связи между этими мирами. Сегодня эта связь может быть организована посредством моделирования виртуальных объектов на основе данных, получаемых с интеллектуальных сенсоров, расположенных на объектах реального мира. Например, в гонках «Формулы 1» поток данных, которые собираются с сотен датчиков, установленных на автомобиле, передается в реальном времени на панель управления и используется для моделирования рабочих характеристик болида в режиме реального времени [2]. Используя эти модели, инженеры

могут вносить корректировки в режим работы автомобиля удаленно, непосредственно в режиме гонки. Такой подход называется «цифровым двойником» (ЦД).

ЦД – это интегрированная мульти-физическая, мульти-масштабная вероятностная симуляция сложного объекта, которая использует наиболее подходящие физические модели, актуальные данные сенсоров и др. для того чтобы получить как можно более достоверное представление соответствующего реального объекта [3]. Реализация моделей ЦД может требовать больших объемов информации и вычислительных ресурсов [4]. Технология *облачных вычислений* позволяет удовлетворять такие требования посредством динамически-масштабируемого предоставления потенциально неограниченного объема вычислительных ресурсов за счет использования технологий виртуализации [5]. *Виртуализация* позволяет разделять физический вычислительный узел на несколько виртуальных машин (ВМ), каждая из которых имеет собственную изолированную операционную систему (ОС) и приложения. Они координируются слоем программного обеспечения под названием *гипервизор*, который обеспечивает поддержку параллельного выполнения нескольких ВМ на одной физической машине [6].

Тем не менее, накладные расходы, связанные с использованием ВМ в облачных вычислениях, могут ограничить эффективность вычислительных ресурсов. Эта проблема может быть решена при помощи технологии контейнеризации [7]. Контейнеризация позволяет запускать независимые контейнеры в виде отдельных процессов непосредственно на ядре базовой ОС, обеспечивая легкую изоляцию процессов внутри контейнеров. Несмотря на сокращение накладных расходов, обеспечиваемое технологией контейнеризации, высокая латентность и возможные перегрузки сетевых каналов не позволяют облачным решениям обеспечить полное соответствие требованиям таких систем как ЦД, чувствительным к временным задержкам и местоположению [8]. Таким образом, облако не может самостоятельно удовлетворить потребности таких приложений.

Технология *туманных вычислений* (fog computing) позволяет решить эту проблему, перемещая часть задач по обработке и хранению данных из облака ближе к *краю сети* (анг. network edge), в так называемые туманные узлы. Системы туманных вычислений поддерживают локальную обработку данных с приемлемой задержкой, обеспечивая предварительную очистку и предобработку данных, перед отправкой в облако [9]. Производительность таких систем критически зависит от способности безопасно и эффективно собирать, передавать и анализировать потоки данных между объектами реального мира и системами обработки данных в режиме реального времени [10].

Процессы, зависящие только от текущего локального состояния, т.е. не от недавнего прошлого или истории всех таких состояний, называются процессами «без сохранения состояния» (stateless). Для удобства читателя в дальнейшем при описании процессов, сервисов и систем, функционирующих в режиме «без сохранения состояния» мы будем называть их «stateless».

Прогностические системы никогда не могут быть реализованы без сохранения состояния, так как для прогнозирования ближайшего будущего они используют не только текущее состояние, но и прошлый опыт [12]. Такие системы принято называть системами «с сохранением состояния» (stateful). Аналогично, в дальнейшем по ходу статьи процессы, сервисы и системы, функционирующие в режиме «с сохранением состояния», мы будем называть «stateful». Stateful-система способна идентифицировать источник входных данных и определить, какие другие данные поступили из того же источника [13]. Разработчики stateful-систем сталкиваются с рядом проблем, включая ограничения масштабируемости и необходимость дополнительных вычислительных затрат на поддержку управления состоянием [14]. Сложность решения этих задач резко возрастает в сложных системах, таких как ЦД, где необходимо обеспечивать stateful-обработку потоков данных между реальными объектами, туманными узлами и облаком.

Сегодня предполагается, что такие сложные распределенные системы должны состоять из набора слабосвязанных независимых компонентов [15]. Такие решения легли в основу «микросервисного» (microservice) архитектурного подхода [16]. Микросервисы – это архитектурный паттерн, который делает акцент на разделении системы на легкие независимые сервисы, каждый из которых отвечает за выполнение отдельной независимой части бизнес-логики приложения [17]. Для обеспечения изоляции микросервисов часто применяются технологии контейнеризации. Однако ограниченная поддержка переносимости состояния в контейнерах приводит к проблемам контейнеризации stateful-сервисов [18]. Кроме того, обработка данных в stateless-режиме является одной из ключевых характеристик микросервисного подхода [16,17]. Stateless-обработка данных позволяет относительно легко добиться таких важных характеристик микросервисных систем, как высокая надежность, высокая доступность, масштабируемость по требованию, балансировка нагрузки [19]. Например, stateless-приложения могут быть развернуты за балансировщиками нагрузки, где их сбои маскируются путем перенаправления трафика на рабочие реплики данного приложения. Но для обеспечения подобных характеристик для stateful-сервисов необходимо использование других подходов [20].

Когда обработка потоков данных требует информации о данных в предыдущих временных сегментах, их реализация в виде stateless-микросервисов становится невозможной. Поддержка сохранения состояния требует применения особых методов для обеспечения высокой надежности и доступности, а приложения такого типа сложнее масштабировать. Решение для управления stateful-сервисами может быть обеспечено посредством платформы обработки потоков данных. Такая система может взять на себя роль «нервной системы» для независимых вычислительных сервисов, в то время как каждый из них выполняет свою часть бизнес-логики.

В этой статье мы представим обзор проблем и требований к решениям по stateful-обработке потоков данных на основе контейнеризированных микросервисов для поддержки развертывания компонентов ЦД на базе туманных вычислений. Кроме того, мы проведем анализ возможности реализации предлагаемого решения с использованием Kafka Streams DSL API[1] для живой миграции контейнеризированных stateful-сервисов обработки потоковых данных.

Статья структурирована следующим образом. В разд. 2 рассматривается проблема обработки данных с сохранением состояния: stateful вычислительной инфраструктуры и stateful-данных. В разд. 3 проводится анализ литературы по теме исследования. В разд. 4 рассматриваются источники данных, и архитектура предлагаемой системы. В разд. 5 рассматривается эксперимент организации живой миграции. В разд. 6 рассматривается эксперимент по распределению вычислительной нагрузки по туманной вычислительной среде. Выводы представлены в разд. 7.

2. Работа с сохранением состояния

Состояние – это любая информация, характеризующая процесс, в течение некоторого внутреннего времени, и может храниться в любом месте иерархии агентов и субагентов, характеризующих процесс [12]. Для выполнения операций с сохранением состояния (stateful-операций) требуется возможность идентификации источника входных данных и определение того, какие еще входные данные были получены из того же источника [13].

Хранение состояния внутри вычислительного сервиса считается узким местом, поэтому состояние должно храниться в отдельном ресурсе [14], например, внешней базе данных, внешней файловой системе, системе кэширования или в промежуточном программном обеспечении для обмена сообщениями.

[1] https://kafka.apache.org

Тем не менее, отдельный ресурс для работы с состоянием приводит к необходимости выделения дополнительных ресурсов, которые могут включать в себя дополнительные вычислительные мощности, память и хранилища данных. Кроме того, такой подход может приводить к проблемам в масштабируемости. В связи с этим, любая stateful-операция сталкивается со сложностью управления состоянием.

Сложность возрастает в условиях туманной вычислительной среды, где важное значение имеет способность живой миграции задач обработки и хранения данных между туманными узлами. *Живой миграцией* называют возможность бесперебойной миграции сервиса между физическими машинами без воздействия на клиентские процессы или приложения. В этом случае вычислительный процесс приостанавливается только на время передачи общего состояния, после чего вычисление возобновляется на целевом узле.

Обеспечение механизма, обеспечивающего правильность обработки, хранения и восстановления состояния, является существенным процессом при реализации stateful-операций. Создание stateful микросервисной системы приводит к целому ряду проблем, наиболее существенными из которых являются предоставление stateful вычислительной инфраструктуры и управление stateful-данными.

2.1 Stateful вычислительная инфраструктура

Сложность проектирования stateful-систем, таких как ЦД, зависит от возможностей базовой вычислительной инфраструктуры. Мы будем называть stateful вычислительной инфраструктурой, такую инфраструктуру виртуализации, которая позволяет хранить и управлять состоянием внутри изолированной вычислительной среды в течение длительного промежутка времени. Технология ВМ обеспечивает возможность создания моментальных снимков и восстановления состояния приложений. Это позволяет осуществлять миграцию ВМ между физическими хостами с минимальным влиянием на запущенные сервисы [21]. Эти технологии применяются во многих платформах управления ВМ, таких как VMware vCenter[1]. Такие снимки позволяют обеспечить миграцию ВМ между вычислительными узлами без существенного прерывания вычислительного процесса и воздействия на их состояние.

К сожалению, такой подход не распространен в технологии контейнеризации. Например, такая миграции не так проста в реализации при использовании платформы Docker[2]. При создании нового контейнера в Docker, новый слой (слой контейнера), доступный для записи, создается поверх слоев, доступных только для чтения (слой изображения), а все изменения, внесенные в контейнер, записываются только на слой контейнера. Методы создания контрольных точек состояния контейнера в настоящее время находятся только в тестовой версии Docker [22]. Это означает, что до сих пор нет непосредственного и оригинального способа организовать живую миграцию докер-контейнеров на другой узел, не потеряв при этом всего состояния.

Существуют несколько методов решения проблемы миграции контейнеров. Например, проект CRIU и его проекты-расширения по-прежнему основаны на экспериментальном режиме Docker[3]. Также CRIU выполняет операцию «PID dance», чтобы восстановить процесс с таким же PID. Эта операция требует привилегированного доступа и обладает низкой производительностью, так как генерирует множество системных вызовов и может привести к возникновению состояния гонки [23]. С другой стороны, контейнеры Linux LXD[4] поддерживают возможность создания снимков и восстановления контейнеров для резервного

[1] https://www.vmware.com/products/vcenter-server.html

[2] https://docs.docker.com/storage/storagedriver/

[3] https://criu.org/Docker

[4] https://ubuntu.com/blog/lxd-2-0-your-first-lxd-container

копирования или миграции. Но, для целей живой миграции, LXD все еще базируется на CRIU[1].

Между тем, некоторые контейнерные платформы, такие как Kubernetes[2], которые изначально строились для поддержки stateless-сервисов, недавно расширили свои модели, включив в них работу с состоянием [12]. Kubernetes предоставляет контроллер StatefulSet для управления stateful-приложениями. Тем не менее, его реализация также ограничена, так как приводит к потере состояния при реализации «rolling-обновлений», сложностям или невозможности применения балансировщиков нагрузки и др. [24]. Ключевой же проблемой в управлении StatefulSet является отсутствие механизмов живой миграции и автоматического создания замены вышедшего из строя StatefulSet [25,26]. Поэтому решение проблемы сохранения состояния при использовании технологии контейнеризации является активной областью исследований. Особенно при обработке stateful-потоков данных, где каждая точка данных может играть решающую роль.

2.2 Stateful-данные

Чтобы организовать управление состоянием, желательно обеспечить постоянное хранение данных о состоянии не только в вычислительных сервисах, но и в других частях системы. Примерами таких систем могут служить внешняя база данных, внешняя файловая система, система кэширования, промежуточное программное обеспечение для обмена сообщениями.

Однако идеальных решений нет, так как любой отдельный ресурс для обработки состояния требует дополнительных серверных ресурсов: дополнительной вычислительной мощности, памяти и места для хранения данных. Например, платформа контейнеризации OpenVZ[3] поддерживает миграцию контейнеров с сохранением состояния с помощью проекта на основе CRIU. OpenVZ использует распределенную систему хранения данных (distributed storage system, DSS) для совместного использования файлов. Однако дополнительные данные, необходимые для репликации на основе DSS, приводят к росту трафика, снижая производительность сети [27]. Таким образом, использование технологий репликации на основе DSS может привести к большим задержкам и ухудшению качества обработки данных на краевых узлах сети.

Проблема усложняется, когда обработка данных ведется не в пакетном, а в потоковом режиме. Потоки данных часто должны обрабатываться последовательно, посредством применения механизмов stateful-обработки временных рядов, таких как скользящее среднее и др. [28].

Для потоковой обработки данных требуются два слоя: *слой хранения* (storage layer, SL) и *слой обработки* (processing layer, PL). SL должен поддерживать быстрый ввод/вывод для больших потоков данных. PL потребляет данные из SL, обрабатывает эти данные и уведомляет SL об обновлении данных. Например, для обеспечения отказоустойчивости, платформа обработки данных Apache Spark[4] поддерживает различные источники входных и выходных данных. Для обеспечения хранения данных она должна быть сконфигурирована с использованием внешней системы хранения, такой как HDFS. Тем не менее, живая миграция для Apache Spark все еще реализуется на базе механизмов миграции ВМ, например, предоставляемых облачной платформой OpenStack [29]. Таким образом, при разработке сложной системы, такой как ЦД, необходимо планировать, как управлять распределением stateful-данных по компонентам

[1] https://lxd.readthedocs.io/en/latest/migration/

[2] https://kubernetes.io/

[3] https://wiki.openvz.org/Virtuozzo_Storage

[4] http://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing

системы и подбирать инструменты обработки данных, обеспечивающие возможности их обработки в режиме сохранения состояния.

3. Обзор литературы

В научном сообществе растет интерес к области управления обработкой данных IoT. Например, авторы [30] предложили распределенный алгоритм отслеживания людей для системы, состоящей из камер видеонаблюдения, подключенных к локальным вычислительным узлам, которые, в свою очередь, подключены к центру обработки данных. Обнаружение людей происходит в центре обработки данных, в то время как локальные вычислительные узлы обеспечивают выполнение задач по предобработке видеопотока.

Авторы [31] используют комбинацию хранилища данных Redis[1] с платформой Apache Storm[2] для обеспечения краткосрочного прогнозирования нагрузки и обнаружения выбросов в потоке данных, генерируемых интеллектуальными датчиками измерения энергии. Для поддержки передачи данных авторы использовали библиотеку обмена сообщениями на основе кольцевого буфера LMAX. Авторы [32] также использовали систему Apache Storm, но для реализации алгоритма статистического контроля под названием «CUmulative SUM» (CUSUM) для выявления аномалий в сериях данных мониторинга окружающей среды. Они использовали Apache ActiveMQ[3] в качестве сервиса обмена сообщениями.

Кроме того, наблюдается взрывной рост платформ и различных решений, предлагаемых для анализа потоков данных. Например, платформа Apache Spark обеспечивает обработку данных как в пакетном, так и в потоковом режиме, поддерживает решения задач машинного обучения. Еще одной платформой, нашедшей свое применение для обработки потоков данных, является Apache Kafka. Apache Kafka является отказоустойчивой платформой распределенной потоковой обработки данных. Apache Kafka хранит данные в *темах* (Kafka topics), которые представляют собой названия категорий, в рамках которых публикуются и потребляются сообщения. Kafka Streams DSL API обеспечивает поддержку обработки локального состояния внутри приложения для таких stateful-операций, как подсчет или агрегирование.

Чтобы отслеживать обновление состояния, Kafka поддерживает реплицируемый журнал изменений для каждого хранилища состояний. Каждое приложение, работающее с Apache Kafka должно иметь уникальный идентификатор для поддержки отслеживания состояния. Однако эти возможности привносят определенные ограничения. Например, операции потребления, сохранения состояния и генерации в Kafka Streams DSL могут выполняться только на одном кластере Kafka. Кроме того, кластер Kafka до сих пор не поддерживает географическую репликацию постоянно хранящихся сообщений в нескольких центрах обработки данных. Это приводит к проблемам применения этой технологии в туманной вычислительной среде. В настоящий момент есть два альтернативных решения: MirrorMaker[4] и Confluent Replicator [33]. MirrorMaker является инструментом для зеркалирования данных между кластерами, но, в дополнение к другим ограничениям, он не дает гарантий того, что конфигурация тем Kafka в реплике будет совпадать с конфигурацией в оригинальном кластере. Также, он не предоставляет UI для мониторинга репликации. Confluent Replicator охватывает многие ограничения MirrorMaker, но является проприетарным решением, требующим платной версии платформы Confluent.

[1] https://redis.io/

[2] https://storm.apache.org/

[3] http://activemq.apache.org/

[4] https://docs.confluent.io/4.0.0/multi-dc/mirrormaker.html

4. Исходные данные и процесс обработки

4.1 Данные и инструменты

Для оценки предлагаемого решения обработки stateful-данных мы использовали реальный набор данных, предоставленный DEBS 2012 Grand Challenge[1], собранный из датчиков, установленных на производственном оборудовании. Задержка между двумя последовательными точками данных составляет около 10 мс. В рамках данного исследования мы остановились на реализации первого запроса по обработке данных. Каждый элемент исходных данных состоит из 66 полей. Первый набор операторов обеспечивает обнаружение изменения состояния во входных полях между последовательными точками исходных данных и выдает их вместе с временными метками изменения состояния. Второй набор операторов решает задачу установления корреляции между изменением состояния датчика и изменением состояния клапана, а также рассчитывает временное расстояние между наступлением изменения состояния и выдает эти данные с временными штампами.

Для автоматизации развертывания Apache Kafka и реестра схем в контейнере мы используем Docker-образ, предоставляемый Lenses[2]. Все микросервисы во всех экспериментах разрабатываются с использованием Java 8 с Kafka Streams DSL API и контейнеризированы с помощью Docker.

4.2 Предлагаемый процесс обработки данных

Мы организуем обработку данных в два этапа. В каждом из них Kafka Streams DSL API автоматически синхронизирует локальное хранилище состояния с промежуточными темами Kafka. Первый этап включает в себя следующие основные шаги:

1) вотребление потока данных из исходной темы;
2) группировка потоков: каждая группа отвечает за свой источник данных;
3) агрегация сгруппированного потока: включает в себя те точки данных, в которых изменилось состояние источника данных;
4) встраивание агрегированных данных в поток результирующих данных: мы создаем новое сообщение с обновленной схемой сообщения, которая включает новое значение изменения состояния вместе с меткой времени;
5) передача результата в виде субпотока следующей группе операторов или отправка его в тему Kafka, если первый этап был реализован в отдельном микросервисе.

Второй этап включает в себя следующие шаги:

1) Если оба этапа обработки выполняются в рамках одного микросервиса, обеспечивается потребление данных от первого этапа обработки; если каждый из этапов обработки реализован в отдельном микросервисе, обеспечивается потребление исходного потока из темы, хранящей результаты первого этапа;
2) Перегруппировка полученных сообщений, которые включают в себя только сообщения об обнаруженном изменении состояния для каждого датчика;
3) Агрегация полученного потока для анализа корреляции между изменением состояния датчиков и изменением состояния клапанов;
4) Встраивание агрегированного значения в поток результатов; создание нового сообщения, которое включает обновленную схему сообщений, состоящую из нового результата анализа корреляции между изменением состояния датчиков и изменением состояния клапана вместе с временной меткой;

[1] http://debs.org/grand-challenges/

[2] https://lenses.io/

5) Отправка результата в тему результатов Kafka.

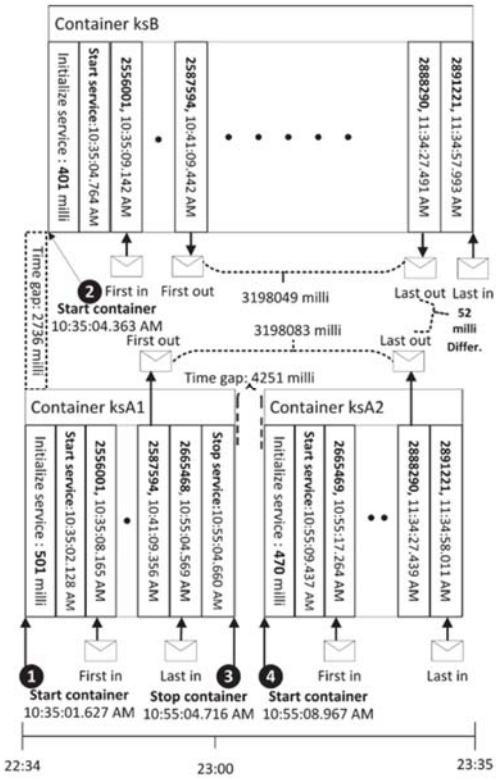

Рис. 1. Эксперимент по живой миграции
Fig. 1. Live migration experiment.

5. Эксперимент по живой миграции

Этот эксперимент проверяет возможность восстановления работы после остановки или выхода из строя контейнера, без влияния на финальные результаты обработки данных. Кроме того, он проверяет возможность использования функции синхронизации локального хранилища состояния с промежуточными темами Kafka в качестве основы для переноса вычислительной задачи в новый контейнер, сохраняя при этом непрерывность результатов из предыдущей точки остановки.

5.1 Методология развертывания и тестирования

Тест начинается с инициализации двух контейнеров (`ksA1`, `ksB`) с одним и тем же микросервисом, но каждый с уникальным идентификатором приложения и различными темами вывода (см. точки 1 и 2 на рис. 1). После этого производится запуск источника данных, который генерирует поток данных в течение одного часа. Через 20 минут завершается работа контейнера `ksA1` (см. точку 3 на рис. 1), а новый контейнер (`ksA2`) запускается с тем же микросервисом, той же темой ввода и вывода, и тем же идентификатором приложения, что и `ksA1` (см. точку 4 на рисунке 1). Таким образом, мы

позволяем `ksA2` повторно использовать предыдущие промежуточные темы `ksA1` для получения состояния `ksA1` в `ksA2`. Через 1 час генерация данных прекращается, и мы оцениваем результаты проведенного эксперимента.

5.2 Оценка результатов эксперимента

Корректность обработки данных составила 100% во всех тестах. Оценка корректности обработки данных была проведена путем сравнения результатов контейнера `ksB` с результатами пары `ksA1` и `ksA2`. Вычисляя разницу во времени между первым и последним итоговыми результатами в каждом тесте, мы можем рассчитать среднее итоговое время, требуемое для завершения обработки всех данных. Среднее общее временя выполнения теста, как при обработке данных посредством одного контейнера `ksB`, так и при прерывании контейнера `ksA1` и запуске контейнера `ksA2`, является очень похожим, с разницей +- 52 мс. Средний промежуток времени между остановкой `ksA1` и запуском `ksA2` составляет 4251 мс. Это время имеет важное значение, поскольку это один из накладных расходов при миграции вычислений из одного контейнера в другой. Среднее время между запуском контейнера и запуском сервиса внутри контейнера составляет 475 мс. Это время – также один из накладных расходов, связанных с запуском контейнеризованного сервиса, а также накладные расходы, связанные с остановкой контейнеров и переносом вычислений в другой контейнер.

6. Эксперимент по распределению вычислительной нагрузки в туманной вычислительной среде

В этом эксперименте мы сравниваем два подхода к организации обработки входных данных (см. рис. 2). Первый подход заключается в реализации обоих этапов процесса обработки данных в одном микросервисе (`ks_total`), размещенным в частном облаке ЮУрГУ, которое располагается рядом с источником данных. Во втором подходе, тот же вычислительный процесс разделяется на два микросервиса (`ks1_susu` и (`ks2_yandex`). `ks1_susu` реализует первый этап обработки данных и развернут в частном облаке в ЮУрГУ в то время как `ks2_yandex` реализует второй этап процесса обработки данных и развернут в публичном облаке компании Яндекс (Яндекс.Облако). Обмен данными между ними организован через два географически разделенных кластера Kafka. Чтобы преодолеть проблемы репликации данных, которые все еще существуют в Kafka, нами был реализован репликатор для организации синхронизации географически разделенных кластеров Kafka. Он разработан на основе концепции микро-потоков работ (анг. Micro-Workflow - MW) [34–36].

На рис. 2 компоненты синей прямоугольной формы в микросервисах `ks_total`, `ks1_susu` и `ks2_yandex` представляют собой вычислительные единицы, представляющие *слой обработки* (PL). Компоненты желтой трубчатой формы (такие как кластеры Kafka и хранилище локальных состояний) представляют собой *слой хранения* (SL). В наших экспериментах SL также распределен: локальное хранилище состояния расположено непосредственно в контейнере, осуществляющем обработку данных. Эксперимент исследует потенциал для повышения параллелизма и возможность развертывания части вычислительной рабочей нагрузки в публичном облаке, в то время как другая часть вычислительной рабочей нагрузки развертывается рядом с источником данных, чтобы обеспечить более быстрое время отклика. Мы сравниваем результаты выполнения обработки данных в рамках системы, состоящей из микросервисов `ks1_susu` и `ks2_yandex` с результатами соответствующих частей обработки данных в `ks_total`.

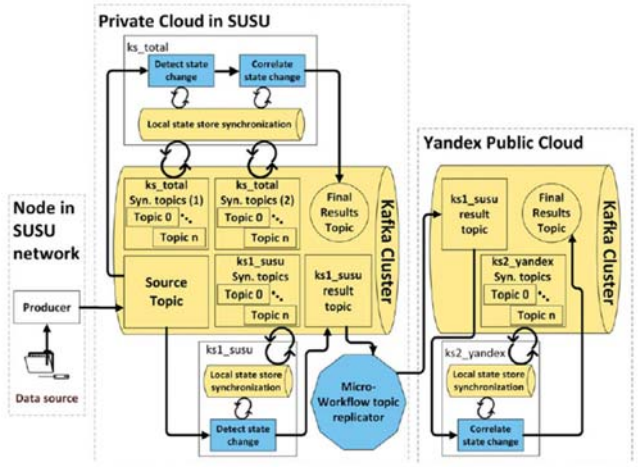

Рис. 2. Детали реализации эксперимента по распределению вычислительной нагрузки
Fig. 2. Implementation details of the computational load distribution experiment.

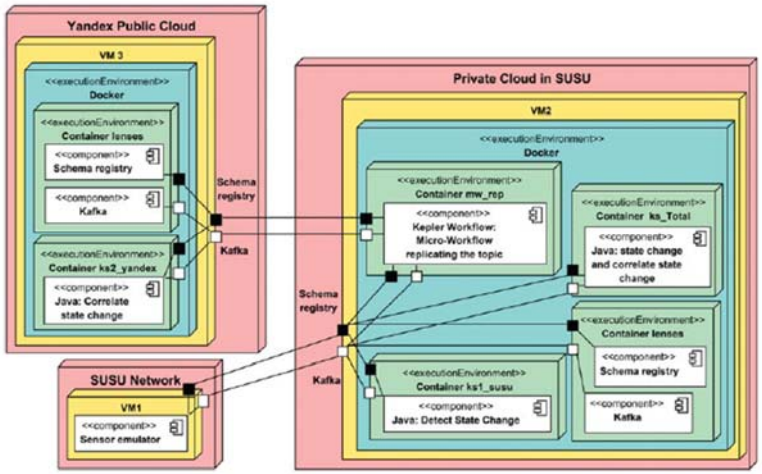

Рис. 3. Размещение компонентов эксперимента по распределению вычислительной нагрузки
Fig. 8. Deployment of the components of the computational load distribution experiment

6.1 Развертывание эксперимента

На рис. 3 показаны детали инфраструктуры, на которой развернут эксперимент.

Эксперимент развернут на трех вычислительных узлах. Узел VM1 (4 ГБ оперативной памяти, двухъядерный процессор Intel Xeon X5680) развернут в сети ЮУрГУ. Здесь находится эмулятор датчика. Узел VM2 (8 ГБ оперативной памяти, восьмиядерный процессор Intel (R) Xeon (R) Gold 6242) развернут в частном облаке в ЮУрГУ. Здесь размещен первый кластер Kafka, контейнер `ks_total`, контейнер `ks1_susu`, а также контейнер `mw_rep`, который включает MW для репликации сообщений из темы в кластере Kafka в кластере VM2 в кластер

Kafka в VM3. Узел VM3 (6 ГБ оперативной памяти, двухъядерный процессор Intel Cascade Lake) развернут в Яндекс.Облаке. Здесь находится второй кластер Kafka и контейнер `ks2_yandex`.

6.2 Оценка результатов эксперимента

Задержка передачи данных между VM1 и VM2 составляет в среднем 2 мс. Задержка между VM2 и VM3 составляет в среднем 30 мс. Продолжительность экспериментов составила 6 часов. К каждом из экспериментов было обработано 1 638 104 исходных сообщения. Промежуток времени между двумя последовательными сообщениями от эмулятора датчика составлял в среднем 14 мс. Сравнение значений всех результатов, полученных `ks1_susu` и `ks2_yandex` с результатами соответствующих частей в `ks_total` показало 100% соответствие результатов обработки данных с точки зрения значений. Оба микросервиса (как `ks1_susu`, так и его соответствующая часть в `ks_total`) получили 1 638 104 входных сообщений и сгенерировали 1 616 результирующих сообщений во время теста.

Зная время первого и последнего результирующего сообщения, мы можем оценить период времени, который потребовался чтобы завершить обработку полученных данных. Рис. 4 показывает, что `ks1_susu` и его соответствующая часть в `ks_total` начать генерацию результатов одновременно, но `ks1_susu` выдает последний результат на 49 мс быстрее.

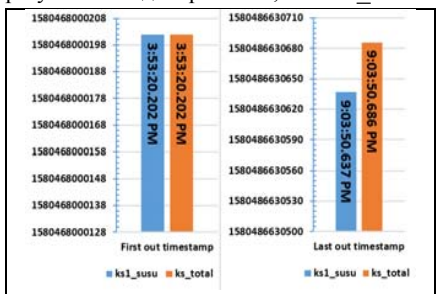

Рис. 4. Временные отметки первых и последних результатов ks1_susu и соответствующей части ks_total
Fig. 4. Time of first and last results of ks1_susu and the corresponding part of ks_total

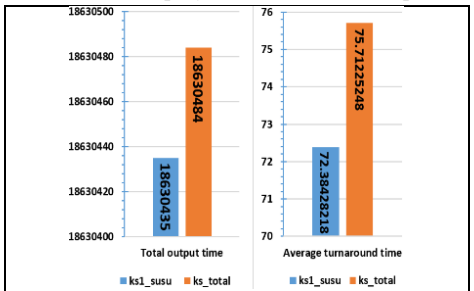

Рис. 5. Итоговое время вывода и время оборота ks1_susu и соответствующая его часть ks_total
Fig. 5. Total output time and turnaround time of ks1_susu and the corresponding part of ks_total

На рис. 5 показано итоговое время вывода и время оборота. *Время оборота* – это средний временной интервал, необходимый для получения одного результирующего сообщения посредством микросервиса. Заметим, что интервал начинается с момента получения второго исходного сообщения, которое участвовало процессе изменения состояния из источника Kafka, и заканчивается в момент отправки результирующего сообщения в соответствующую тему Kafka (в случае `ks1_susu`) или в следующий этап рабочего процесса обработки данных (в случае `ks_total`). `ks1_susu` завершил работу быстрее, чем его соответствующая часть в `ks_total`. Кроме того, среднее время оборота `ks1_susu` было меньше, чем среднее время оборота его соответствующей части в `ks_total`, примерно на 3 мс.

Как `ks2_yandex`, так и его соответствующая часть в `ks_total` получили 1 616 исходных сообщения и сгенерировали 816 результирующих сообщения во время теста. Данные на выходе из `ks1_susu` являются потоком данных, получаемых `ks2_yandex`.

Результаты обработки `ks1_susu` передаются в тему, расположенную в первом кластере Kafka в частном облаке в ЮУрГУ. Это сообщение затем реплицируется репликатором на базе MW в тему во втором кластере Kafka в Яндекс.Облаке, а затем потребляется `ks2_yandex`

для реализации этапа обработки данных «Установление корреляции». В рамках же `ks_total`, входные данные для этого шага обработки являются данными результатов первого шага рабочего процесса, реализованного в том же `ks_total` (см. рис. 2).

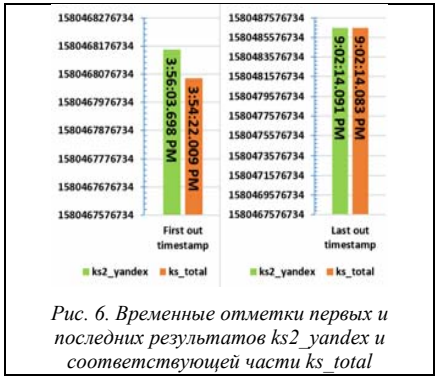

Рис. 6. Временные отметки первых и последних результатов ks2_yandex и соответствующей части ks_total

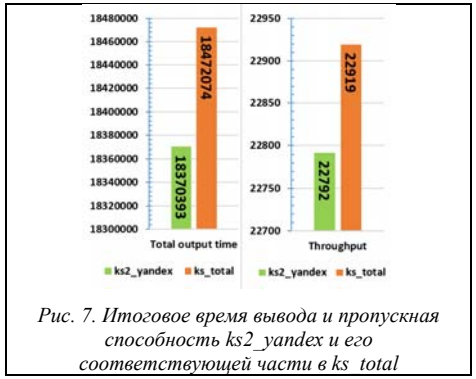

Рис. 7. Итоговое время вывода и пропускная способность ks2_yandex и его соответствующей части в ks_total

При анализе времени получения результатов надо учитывать, что результаты `ks2_yandex` включают дополнительную задержку, связанную с передачей данных между VM2, развернутом в частном облаке в ЮУрГУ, и VM3, развернутом в Яндекс.Облаке. Тем не менее, время задержки постепенно сокращается во время экспериментов, см. рис. 6. Сокращение времени задержки можно объяснить следующим образом: `ks1_susu` генерирует результаты быстрее, чем соответствующая часть `ks_total` (см. рис. 5). Кроме того, `ks2_yandex` генерирует результаты быстрее, чем его соответствующая часть в `ks_total`. Итоговое время на получение всех результатов в `ks2_yandex` составляет на 101 681 мс меньше, чем итоговое время, затраченное на получение результатов в соответствующей части в `ks_total`. Таким образом, `ks2_yandex` имеет более высокую пропускную скорость по сравнению с соответствующей частью в `ks_total` на 127 мс (см. рис. 7).

7. Выводы

Наше исследование показывает важность двух факторов, влияющих на построение обработки потоков данных с сохранением состояния в таких сложных системах, как ЦД: stateful вычислительная инфраструктура и stateful-данные. Stateful вычислительная инфраструктура упрощает сложность и характер вычислительной деятельности за счет способности инфраструктуры хранить и управлять данными внутри изолированной вычислительной системы в течение длительных промежутков времени. Тем не менее, важно планировать, как управлять stateful-данными при реализации микросервисной архитектуры и выбирать инструменты обработки данных, которые обеспечивают сохранение состояния. Кроме того, если система обработки потоков данных не поддерживает репликацию сообщений в географически разделенных центрах обработки данных, это создает проблемы с точки зрения надежности, долговечности и доступности данных.

Результаты нашего эксперимента показывают, что подход, основанный на создании локальных хранилищ состояния в привязке к приложениям на слое обработки и синхронизации промежуточных данных со слоем хранения (как, например Kafka Stream DSL API в наших экспериментах) может быть использован для поддержки живой миграции контейнеризованных приложений с сохранением состояния в туманной вычислительной среде с минимальными накладными расходами (за исключением повышения временной задержки). Если состояние хранится в слое хранения, на задержку влияют два основных

фактора. Во-первых, расстояние между приложением и слоем хранения. Во-вторых, это объем трафика, который генерируется при обмене промежуточными данными между приложением и слоем хранения. Кроме того, результат наших экспериментов показывает, что распределение обработки потока данных на изолированные сервисы повышает параллелизацию.

В качестве направления дальнейших исследований мы планируем развивать MW подход к обработке потоков данных. Важными темами для дальнейших исследований являются исследование репликаторов и подходов обработки потоков данных в географически-разделенных центрах обработки данных для поддержки развертывания компонентов ЦД в туманной вычислительной среде.

Список литературы / References

[1]. A. Tchernykh, M. Babenko, N. Chervyakov et al. Scalable Data Storage Design for Non-Stationary IoT Environment with Adaptive Security and Reliability. IEEE Internet of Things Journal, vol. 7, no. 10, 2020, pp. 10171-10188.

[2]. G. E. Modoni, M. Sacco, W. Terkaj. A Telemetry-driven Approach to Simulate Data-intensive Manufacturing Processes. Procedia CIRP, vol. 57, 2016, pp. 281-285.

[3]. E. H. Glaessgen and D. D. S. Stargel. The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles. In Proc. of the 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference - Special Session on the Digital Twin, 2012, pp. 1-14.

[4]. M. Grieves and J. Vickers. Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems. In Transdisciplinary Perspectives on Complex Systems, Springer, 2017, pp. 85–113.

[5]. Q. Zhang, L. Cheng, R. Boutaba. Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications, vol. 1, no. 1, 2010, pp. 7-18.

[6]. Б.М. Шабанов, О.И. Самоваров. Принципы построения межведомственного центра коллективного пользования общего назначения в модели программно-определяемого ЦОД. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 7-24. DOI: 10.15514/ISPRAS-2018-30(6)-1 / B.M. Shabanov and O.I. Samovarov. Building the Software-Defined Data Center. Programming and Computer Software, vol. 45, no. 8, 2019, pp. 458-466 (in Russian).

[7]. G. I. Radchenko, A. B. A. Alaasam, A. N. Tchernykh. Comparative Analysis of Virtualization Methods in Big Data Processing. Supercomputing Frontiers and Innovations, vol. 6, no. 1, 2019, pp. 48-79.

[8]. J. Luo, L. Yin, J. Hu et al. Container-based fog computing architecture and energy-balancing scheduling algorithm for energy IoT. Future Generation Computer Systems, vol. 97, 2019, pp. 50-60.

[9]. M. Aazam, S. Zeadally, K. A. Harras. Deploying Fog Computing in Industrial Internet of Things and Industry 4.0. IEEE Transactions on Industrial Informatics, vol. 14, no. 10, 2018, pp. 4674-4682.

[10]. S. Singh, A. Angrish, J. Barkley et al. Streaming Machine Generated Data to Enable a Third-Party Ecosystem of Digital Manufacturing Apps. Procedia Manufacturing, vol. 10, 2017, pp. 1020-1030.

[11]. Y. Qamsane, C. Chen, E. C. Balta et al. A Unified Digital Twin Framework for Real-time Monitoring and Evaluation of Smart Manufacturing Systems. In Proc. of the 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), 2019, pp. 1394-1401.

[12]. M. Burgess. Locality, Statefulness, Causality in Distributed Information Systems Concerning the Scale Dependence Of System Promises. arXiv, 2019, available: http://arxiv.org/abs/1909.09357.

[13]. C. Peiffer and I. L'Heureux. System and method for maintaining statefulness during client-server interactions. United States Patent. US8346848B2, 2013.

[14]. M. Naseri and A. Towhidi. Stateful Web Services: A Missing Point in Web Service Standards. In In Proc. of the International MultiConference of Engineers and Computer Scientists, 2007, pp. 993-997.

[15]. R. Lichtenthaler. Model-driven software migration towards fine-grained cloud architectures. CEUR Workshop Proceedings, vol. 2339, 2019, pp. 35-38.

[16]. S. Newman. Building Microservices: Designing Fine-Grained System. O'Reilly Media, 2015, 280 p.

[17]. James Lewis and Martin Fowler. Microservices. 2014. Available at: https://martinfowler.com/articles/microservices.html, accessed Jan. 11, 2019.

[18]. The Microservice Revolution: Containerized Applications, Data and All, Available at: https://www.infoq.com/articles/microservices-revolution, accessed Dec. 10, 2019.

[19]. C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. Cloud Computing Patterns: Fundamentals to Design, Build, Manage Cloud Applications. Vienna: Springer Vienna, 2014.

[20]. W. Li and A. Kanso. Comparing containers versus virtual machines for achieving high availability. In Proc. of the 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 353-358.
[21]. C. Clark, K. Fraser, S. Hand et al. Live Migration of Virtual Machines. In Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation, vol. 2, 2005, pp. 273-286.
[22]. docker checkpoint | Docker Documentation, Available at: https://docs.docker.com/engine/reference/commandline/checkpoint, accessed Dec. 11, 2019.
[23]. A. Reber. CRIU and the PID dance. In Proc. of the Linux Plumbers Conference, 2019, pp. 1-4.
[24]. StatefulSets – Kubernetes, Available at: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#limitations, accessed Dec. 18, 2019.
[25]. L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, F. Khendek. Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes. In Proc. of the 19th IEEE International Conference on Software Quality, Reliability and Security, 2019, pp. 176–185.
[26]. H. Loo, A. Yeo, K. Yip, T. Liu. Live Pod Migration in Kubernetes. University of British Columbia, Vancouver, Canada. [Online]. Available at : https://www.cs.ubc.ca/~bestchai/teaching/cs416_2017w2/project2/project_m6r8_s8u8_v5v8_y6x8_proposal.pdf
[27]. H. Ohtsuji and O. Tatebe. Network-Based Data Processing Architecture for Reliable and High-Performance Distributed Storage System. Lecture Notes in Computer Science, vol. 9523, 2015, pp. 16–26.
[28]. A. B. A. Alaasam, G. Radchenko, A. Tchernykh. Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL. In Proc. of the International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), 2019, pp. 0804–0809.
[29]. A. Raddaoui, A. Settle, J. Garbutt, S. Singh. High Availability of Live Migration. openstack.org, 2017. [Online]. Available at: http://superuser.openstack.org/wp-content/uploads/2017/06/ha-livemigrate-whitepaper.pdf, accessed Dec. 11, 2019.
[30]. Д.А. Купляков, Е.В. Шальнов, В.С. Конушин, А.С. Конушин. Распределенный алгоритм сопровождения для подсчета людей в видео. Программирование, том. 45, no. 4, стр. 28-35. / D.A. Kuplyakov, E.V. Shalnov, V.S. Konushin, A.S. Konushin. A Distributed Tracking Algorithm for Counting People in Video. Programming and Computer Software, vol. 45, no. 4, 2019, pp. 163–170.
[31]. A. Sunderrajan, H. Aydt, A. Knoll. DEBS Grand Challenge : Real time Load Prediction and Outliers Detection using STORM. In Proc. of the 8th ACM International Conference on Distributed Event-Based Systems, 2014, pp. 294–297.
[32]. S. Trilles, S. Schade, O. Belmonte, J. Huerta. Real-Time Anomaly Detection from Environmental Data Streams. Lecture Notes in Geoinformation and Cartography, vol. 217, 2015, pp. 125–144.
[33]. Apache Kafka's MirrorMaker – Confluent Platform, Available at: https://docs.confluent.io/4.0.0/multi-dc/mirrormaker.html, accessed Apr. 15, 2020.
[34]. G. Radchenko, A. Alaasam, A. Tchernykh. Micro-Workflows: Kafka and Kepler Fusion to Support Digital Twins of Industrial Processes. In Proc. of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 83–88.
[35]. А.Б.А. Алаасам, Г.И. Радченко, А. Н. Черных. Микро-потоки работ: сочетание потоков работ и потоковой обработки данных для поддержки цифровых двойников технологических процессов. Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика, том 8, no. 4, 2019 г., стр. 100-116 / A. B. A. Alaasam, G. Radchenko, A. Tchernykh. Micro-Workflows: A Combination of Workflows and Data Streaming to Support Digital Twins of Production Processes. Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering, vol. 8, no. 4, 2019, pp. 100–116, Nov. 2019 (in Russian).
[36]. A.B.A. Alaasam, G. Radchenko, A. Tchernykh et al. Scientific Micro-Workflows : Where Event-Driven Approach Meets Workflows to Support Digital Twins. In Proc. of the International Conference RuSCDays'18 – Russian Supercomputing Days, vol. 1, 2018, pp. 489–495.

Информация об авторах / Information about authors

Амир Басим Абдуламир АЛААСАМ является аспирантом кафедры системного программирования. Область его научных интересов включает в себя технологии

распределенных вычислительных систем, включая методы поточной обработки данных, облачные и туманные вычисления, системы потоков работ.

Ameer Basim Abdulameer ALAASAM is a postgraduate student of the Department of System Programming. The area of his scientific interests includes technologies of distributed computing systems, including methods of stream data processing, cloud and fog computing, workflow systems.

Глеб Игоревич РАДЧЕНКО – кандидат физико-математических наук, доцент, директор Высшей школы электроники и компьютерных наук, заведующий кафедрой Электронных вычислительных машин. Область научных интересов: технологий распределенной обработки данных, облачные и туманные вычислительные системы, проблемно-ориентированные распределенные вычислительные системы.

Gleb Igorevich RADCHENKO – Candidate of Physical and Mathematical Sciences, Associate Professor, Director of the School of Electronic Engineering and Computer Science, Head of the Department of Computers. Research interests: distributed data processing technologies, cloud and fog computing systems, problem-oriented distributed computing systems.

Андрей Николаевич ЧЕРНЫХ получил степень кандидата наук в Институте точной механики и вычислительной техники РАН. Он является профессором CICESE. В научном плане его интересуют многоцелевая оптимизация распределения ресурсов в облачной среде, проблемы безопасности, планирования, эвристики и метаэвристики, интернет вещей и т.д.

Andrei Nikolaevitch TCHERNYKH received his PhD degree at the Institute of Precision Mechanics and Computer Engineering of the Russian Academy of Sciences. He is holding a full professor position in computer science at CICESE. He is interesting in grid and cloud research addressing multiobjective resource optimization, both, theoretical and experimental, security, uncertainty, scheduling, heuristics and meta-heuristics, adaptive resource allocation, and Internet of Things.

Хосе Луис ГОНСАЛЕС-КОМПЕАН получил степень кандидата наук в области компьютерной архитектуры в Политехническом университете Каталонии в Барселоне. В настоящее время он является штатным профессором-исследователем в Центре перспективных исследований. Его направления исследований: облачные вычисления и контейнерные системы хранения, лингвистические архивные системы, безопасные и федеративные сети хранения.

José Luis GONZÁLEZ-COMPEÁN received his PhD degree in Computer architecture from UPC Universitat Politècnica de Catalunya, Barcelona. Currently he is full time professor researcher at Center of Research and Advanced Studies. His research lines: cloud computing and containerized storage systems, linguistic archival systems, secure and federated storage networks.