Федеральное государственное автономное образовательное учреждение
высшего образования «Южно-Уральский государственный университет
(национальный исследовательский университет)»

На правах рукописи

АЛААСАМ Амир Басим Абдуламир

МОДЕЛИ, МЕТОДЫ И АЛГОРИТМЫ ОБРАБОТКИ ПОТОКОВ ДАННЫХ В ТУМАННЫХ ВЫЧИСЛИТЕЛЬНЫХ СРЕДАХ

Специальность 05.13.11 – Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
РАДЧЕНКО Глеб Игоревич,
кандидат физ.-мат. наук, доцент

Челябинск – 2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность темы исследования

Актуальность темы диссертационного исследования основывается на следующих основных факторах:

1) экспоненциальный рост устройств и систем интернета вещей;
2) 4-я индустриальная революция и развитие технологий, связанных с Цифровыми двойниками;
3) естественные ограничения применимости модели облачных вычислений, связанные с латентностью при обработке потоков данных;
4) необходимость разработки и исследования новых подходов и моделей организации вычислительного процесса в гетерогенных распределенных вычислительных средах, обеспечивающих обработку потоков данных.

Рассмотрим эти факторы более подробно.

Технологии *интернета вещей (Internet of Things, IoT)* сегодня переживают стадию стремительного роста. По данным Allied Market Research, мировой рынок датчиков оценивался в $138 965 млн в 2017 году и, по прогнозам, достигнет $287 002 млн к 2025 году [127]. Специалисты корпорации Ericsson подсчитали, что по состоянию на 2021 год, в мире насчитывается более 28 миллиардов подключенных к Интернету устройств, что составляет более 3 устройств на каждого жителя земли [26]. Из-за широкого распространения использования технологий IoT, в настоящее время физический и цифровой миры стали взаимосвязанными, что послужило мотивом для установления взаимосвязи между этими двумя мирами посредством телеметрии, поддерживаемой моделированием. Например, в автогонках Формулы-1 поток данных, собранный с сотен датчиков, установленных на автомобиле, и передаваемый на пульт технического обслуживания, служит источником

данных для моделирования работы автомобиля в реальном времени [71]. Используя эти модели, инженеры могут вносить корректировки в режим работы автомобиля удаленно, непосредственно в режиме гонки. Использование таких подходов в индустриальной сфере называется *индустриальным интернетом вещей (Industrial Internet of Things, IIoT)* целью которого является создание *умной индустрии (Smart Industry)* или *Индустрии 4.0 (Industry 4.0)*, которая интегрирует IoT с производственными технологиями для создания взаимосвязанного производственного предприятия, которое анализирует информацию для осуществления интеллектуальных действий в физическом мире [80].

Важным приложением умной индустрии является так называемый *цифровой двойник (Digital Twin, DT)*. DT представляет собой систему, состоящую из трех основных компонентов: физический объект в реальном мире, его виртуальное представление в виртуальном мире, а также потоки данных и управления, которые объединяют реальные и виртуальные компоненты [34]. В отличие от традиционного моделирования, виртуальное представление в DT постоянно обновляется с учетом состояния обслуживания и производительности на протяжении всего жизненного цикла физического объекта [65]. Для создания DT технологических процессов и систем применяются системы математических моделей и методов, таких как интеллектуальный анализ данных, метод конечных элементов и т.д. [58]. Каждый из таких методов предъявляет особые требования к необходимым вычислительным ресурсам. Например, методы интеллектуального анализа данных требуют вычислительных ресурсов, предоставляющих существенные объемы хранения данных [118], в то время как модели, использующие метод конечных элементов, требуют высокопроизводительных вычислительных систем (или суперкомпьютеров) [114].

Из-за необходимости сбора, передачи и анализа потоков данных от систем DT в режимах, близких к реальному времени, для настройки и

актуализации их виртуального состояния, применение облачных технологий не позволяет обеспечить требуемые характеристики предоставляемых вычислительных ресурсов с точки зрения времени задержки и местоположению сервисов обработки данных [20,65]. Возможным решением этой проблемы может служить применение модели *туманных вычислений*, которая расширяет концепцию облачных вычислений, предоставляя вычислительные ресурсы ближе к источникам данных [48]. Туманные вычисления — это многоуровневая модель, обеспечивающая повсеместный доступ к общему континууму масштабируемых вычислительных ресурсов и поддерживающая развертывание распределенных приложений и сервисов с учетом латентности [49]. Хотя промышленные данные часто являются неструктурированными, их можно уточнить и предварительно обработать локально на уровне туманных вычислений перед отправкой на облачный уровень для дальнейшей обработки [1]. Концепция туманных вычислений позволяет перенести часть задач по обработке и хранению данных из облака на туманные узлы на границе сети для снижения задержки.

Для организации эффективной обработки данных в ограничениях, накладываемых, с одной стороны, особенностями предметной области IoT и DT, а с другой стороны, возможностями и особенностями архитектуры туманных вычислительных систем, может быть применен ряд существующих подходов. *Событийно-управляемая архитектура (Event-Driven Architecture – EDA)* наиболее адаптирована к этому типу приложений. EDA — это системная архитектура, состоящая из слабосвязанных, компонентов обработки событий, которые принимают и обрабатывают события одновременно [68]. EDA, по своей природе, является экстремально слабосвязанной и высоко распределенной архитектурой программных систем [37]. С другой стороны, для решения задачи обработки данных в DT применяется концепция *научных потоков работ (Scientific Workflow – SWF)*. *Научным потоком работ* называют набор взаимосвязанных вычислительных задач и задач по обработке

данных, направленных на достижение конкретной цели, в частности на проведение вычислительного эксперимента [59]. Тем не менее, сегодня можно выделить несколько ключевых проблем, связанных с использованием SWF для обработки потоков данных IoT и DT. Во-первых, SWF не ориентированы на обработку потоков данных [15]. SWF исторически ориентированы на исполнение вычислительных задач в пакетном виде, где набор исходных данных собирается и подается в SWF в виде пакета, который и обрабатывается в рамках соответствующего потока работ [45]. Кроме того, действия SWF могут генерировать большое количество промежуточных данных в течение жизненного цикла SWF [122]. В такой сильносвязанной архитектуре, интенсивная передача данных между действиями SWF может вызвать значительное затруднение в процессе выполнения [72].

Анализ исследований, направленных на декомпозицию потоков работ на «под-потоки» показывает, что при решении этой задачи, авторы работ оставляют сильные связи между под-потоками; потоки работ реализуются в формате пакетной обработки данных, что не позволяет применить эти решения в контексте событийно-управляемой архитектуры для поддержки систем IoT в туманных вычислительных средах [4,58,103,113]. Также, большинство исследователей не фокусируются на потребностях туманных вычислений, которые включают необходимость географического распределения, а также необходимость слабосвязанной архитектуры не только между данными и обработкой, но и в самом слое обработки, где каждый вычислительный объект может быть реализован как независимый сервис [21,41,101]. В связи этим, разработка моделей, методов и алгоритмов обработки потоков данных в туманных вычислительных средах является актуальной задачей.

Степень разработанности темы

Фундаментом современных подходов к организации предоставления распределенных вычислительных сервисов является идея коммунальных

вычислений (Utility computing) предложенная предложена Дж. Маккарти (John McCarthy) [31] и Т. Курцом (Thomas Kurtz) [28]. Развитие данной концепции привело к появлению подхода метавычислений (metacomputing) позднее трансформировавшегося в концепцию грид-вычислений (Grid computing). Важный вклад в развитие этих подходов внесли такие ученые, как Л. Смарр (Larry Smarr) [97], М. Мутка (Matt Mutka) и М. Ливны (Miron W. Livny) [61,73], Ян Фостер (Ian Foster) [19], А. Штрайт (Achim Streit) [99], Д. Андерсон (David P. Anderson) [12]. Развитие систем виртуализации и контейнеризации, привело к формированию концепции облачных вычислений, которая стала стандартом де-факто в организации предоставления вычислительных ресурсов по требованию [119]. Развитием данной концепции стала архитектура туманных вычислений, призванная решить задачу минимизации латентности, вместе с предоставлением унифицированного континуума вычислительных возможностей. Существенный вклад в развитие данного подхода внесли такие ученые, как П. Беллависта (Paolo Bellavista) [18], А. Дэви (Alan Davy) [105], М. Аазам (Mohammad Aazam), Ш. Зеадалли (Sherali Zeadally) и Х.А. Харрас (Khaled A. Harras) [1].

Важным аспектом исследований в этой области является решение задачи управления вычислительными задачами, и обработкой данных в распределенных вычислительных системах. Модель научных потоков работ (Scientific Workflow) сегодня представляет собой основную модель, ориентированную на решение подобных задач. Важнейшие работы в области, связанной с проектированием, планированием и выполнением потоков работ сегодня выполняются группами ученых под руководством Е. Дильман (Ewa Deelman) [24,83], Ц. Ванга (Jianwu Wang) [113], Т. Фахрингера (Thomas Fahringer) [27,84], Р. Сакелляриу (Rizos Sakellariou) [90], И. Алтинтаса (Ilkay Altintas) [82], П. Корамбатха (Prakashan Korambath) [57,58], Б. Людешера (Bertram Ludäscher) [63], А.Н.Черных [106].

В области обработки потоков данных можно отметить результаты работы научных групп под руководством таких ученых, как А. Сундерраджан (Abhinav Sunderrajan) [101], А. Антони (Aleksandar Antoni) [14], С. Триллес (Sergio Trilles) [108], О. Карвальо (Otávio Carvalho) [21], С. Хааг (Sebastian Haag) [41], Д. Шейбмайр (Jim Scheibmeir) [92].

Среди российских ученых существенный вклад в решение задач разработки моделей распределенных вычислительных систем, и обработки потоков данных был сделан в работах А.В. Бухановского, С.В. Ковальчука [56], Д.А. Насонова [98], О.В. Сухорослова [100], В. Ильина [107], Вл.В. Воеводина [112,126], и некоторых других.

Цель и задачи исследования

Целью исследования является разработка новой концепции организации потоков работ, включая математическую модель, методы и алгоритмы, позволяющей организовать эффективную обработку потоков данных в туманных вычислительных средах. Для достижения этой цели необходимо решить следующие задачи:

1) проанализировать известные концепции и принципы обработки потоковых данных, используемые для реализации приложений в туманных вычислительных средах;
2) разработать новую математическую модель организации потоков работ, ориентированную на эффективную обработку потоков данных в туманных вычислительных средах;
3) разработать алгоритм преобразования монолитных приложений потоков работ в наборы независимых потоков работ, поддерживающих поточную обработку данных;
4) разработать комплекс программных компонентов и утилит для поддержки обработки потоков данных в туманных вычислительных средах посредством потоков работ;

5) провести вычислительные эксперименты для оценки эффективности предложенной концепции и разработанного программного обеспечения.

Научная новизна

Новизна работы заключается в том, что разработана новая концепция организации потоков работ, получившая название «концепция микро-потоков работ», включающая в себя модель, методы и алгоритмы, позволяющие обеспечить эффективную обработку потоков данных в туманных вычислительных средах с применением концепции потоков работ, позволяющая на порядки уменьшить время задержки получения результата при обработке потоков данных.

Теоретическая и практическая значимость работы

Теоретическая значимость работы заключается в том, что разработанная концепция микро-потоков работ, обеспечивающая организацию обработки потоков данных на базе потоков работ, включает в себя формальную модель организации обработки данных и алгоритм организации рефакторинга монолитных потоков работ в наборы слабосвязанных микро-потоков работ. *Практическая значимость* работы состоит в разработке набора программных акторов и набора утилит, обеспечивающих интеграцию системы управления потоками работ Kepler и платформы обработки потоков данных Apache Kafka для реализации обработки потоков данных в виде микро-потоков работ.

Методология и методы исследования

Методологической основой диссертационного исследования являются теория множеств и теория графов. При разработке программных компонентов применялись методы объектно-ориентированного проектирования и язык

UML. Для программной реализации разработанных подходов были использованы методы объектно-ориентированного проектирования, язык Java, платформа контейнеризации приложений Docker, система управления потоками работ Kepler и платформа обработки потоков данных Apache Kafka.

Положения, выносимые на защиту

На защиту выносятся следующие новые научные результаты:

1. Разработана новая концепция микро-потоков работ, ориентированная на организацию обработки данных в туманных вычислительных средах, позволяющая значительно уменьшить время задержки получения результата при обработке потоков данных.
2. Разработан алгоритм рефакторинга монолитных приложений потоков работ в наборы независимых микро-потоков работ.
3. Выполнены проектирование и реализация комплекса вычислительных акторов и программных утилит для поддержки функционирования микро-потоков работ на базе платформы управления потоками работ Kepler и платформы обработки потоков данных Apache Kafka.
4. С использованием разработанного комплекса программных компонентов созданы микро-потоки работ, обеспечивающие поддержку типовых задач обработки данных, на базе которых проведены вычислительные эксперименты, подтверждающие эффективность предложенных подходов.

Степень достоверности результатов

Результаты исследования подтверждаются данными вычислительных экспериментов, выполненных в соответствии с общепринятыми стандартами.

Апробация результатов исследования

Основные положения диссертационной работы, разработанные модели, методы, алгоритмы и результаты вычислительных экспериментов докладывались автором на следующих международных и всероссийских научных конференциях и семинарах:

1. RuSCDays 2018: Международная конференция «Суперкомпьютерные дни в России» (24-25 Сентября 2018 г., Москва).
2. UCC'2018: 2018 IEEE/ACM International Conference on Utility and Cloud Computing (17-20 Декабря 2018, Цюрих, Швейцария).
3. SIBIRCON'2019: 2019 International Multi-Conference on Engineering, Computer and Information Sciences (21-27 Октября 2019 г., Екатеринбург).

Публикации соискателя по теме диссертации

Основные результаты диссертации опубликованы в следующих научных работах.

Публикации в журналах из списка ВАК

1. Alaasam, A.B.A. Refactoring the Monolith Workflow into Independent Micro-Workflows to Support Stream Processing / A.B.A. Alaasam, G. Radchenko, A. Tchernykh // Programming and Computer Software. –2021. – Vol. 47, No. 8. –P. 591-600. DOI: 10.1134/S0361768821080077. – ***также индексируется в Web of Science и Scopus***.
2. Alaasam, A.B.A. Analytic Study of Containerizing Stateful Stream Processing as Microservice to Support Digital Twins in Fog Computing / A.B.A. Alaasam, G. Radchenko, A. Tchernykh, J. L. González Compeán // Programming and Computer Software. –2020. –Vol. 46, No. 8. –P. 511–525. DOI:10.1134/S0361768820080083. – ***также индексируется в Web of Science и Scopus***.

3. Alaasam, A.B.A. Micro-Workflows Data Stream Processing Model for Industrial Internet of Things / A.B.A. Alaasam, G. Radchenko, A. Tchernykh // Supercomputing Frontiers and Innovations. –2021. –Vol. 8, No. 1. –P. 82–98. DOI:10.14529/jsfi210106. – ***также индексируется в Scopus***.
4. Radchenko, G. Comparative Analysis of Virtualization Methods in Big Data Processing / G. Radchenko, A.B.A. Alaasam, A. Tchernykh // Super-computing Frontiers and Innovations. –2019. –Vol. 6, No. 1. –P. 48–79. DOI:10.14529/jsfi190107. – ***также индексируется в Scopus***.
5. Алаасам, А.Б.А. Цифровые двойники в туманных вычислениях: организация обработки данных с сохранением состояния на базе микропотоков работ / А.Б.А. Алаасам, Г. И. Радченко, А. Н. Черных, Х.Л. Гонсалес-Компеан // Труды Института системного программирования РАН. –2021. – Т. 33, № 1. – С. 65–80. DOI:10.15514/ISPRAS-2021-33(1)-5.
6. Алаасам, А.Б.А. Микро-потоки работ: сочетание потоков работ и потоковой обработки данных для поддержки цифровых двойников технологических процессов / А.Б.А. Алаасам, Г. И. Радченко, А. Н. Черных // Вестник ЮУрГУ. Серия Вычислительная математика и информатика. –2019. – Т. 8, № 4. – С. 100–116. DOI:10.14529/cmse190407.

Публикация, индексируемая в Web of Science

(не включенная в список ВАК)

7. Radchenko, G. Micro-Workflows: Kafka and Kepler Fusion to Support Digital Twins of Industrial Processes / G. Radchenko, A.B.A. Alaasam, A. Tchernykh // 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). –Zurich, Switzerland: – IEEE, –2018. No. 18. –P. 83–88. DOI:10.1109/UCC-Companion.2018.00039. – ***также индексируется в Scopus***.

Публикации, индексируемые в Scopus

(не включенные в список ВАК и/или Web of Science)

8. Alaasam, A.B.A. The Challenges and Prerequisites of Data Stream Processing in Fog Environment for Digital Twin in Smart Industry / A.B.A. Alaasam // International Journal of Interactive Mobile Technologies. –2021. –Vol. 15, No. 15. –P. 126–139. DOI:10.3991/ijim.v15i15.24181.

9. Alaasam, A.B.A. Stateful Stream Processing for Digital Twins: Micro-service Based Kafka Stream DSL / A.B.A. Alaasam, G. Radchenko, A. Tchernykh // 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). –IEEE, –2019. –P. 0804–0809. DOI:10.1109/SIBIRCON48586.2019.8958367.

Публикация, индексируемая в РИНЦ

10. Alaasam, A.B.A. Scientific Micro-Workflows: Where Event-Driven Approach Meets Workflows to Support Digital Twins / A.B.A. Alaasam, G. Radchenko, A. Tchernykh, K. Borodulin, A. Podkorytov // Суперкомпьютерные дни в Росси: труды международной конференции. –2018. –С. 489–495.

Свидетельства о регистрации программ для ЭВМ

11. Алаасам, А.Б.А., Радченко Г.И. Свидетельство Роспатента о государственной регистрации программы для ЭВМ "Комплекс акторов для поддержки концепции Micro-Workflow на платформе Kepler" № 2021661464 от 12.07.2021.

Публикации. По теме диссертации опубликовано 11 работ. Из них: 5 работ опубликовано в журналах индексируемых в Scopus и Web of Science, и 6 работ – в журналах, включенных ВАК в перечень изданий, в которых должны быть опубликованы основные результаты диссертаций на соискание ученой степени доктора и кандидата наук.

Личный вклад автора. Все результаты, представленные в диссертационной работе, получены автором лично. Содержание диссертации и

основные положения, выносимые на защиту, соответствуют персональному вкладу автора в работах, опубликованных в соавторстве. В работе [1] Г.И. Радченко принадлежит раздел 1 (введение, стр. 591–592), А.Н. Черных принадлежит раздел 2 (обзор текущего состояния исследований, стр. 592), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 592–600). В работе [2] Г.И. Радченко принадлежит раздел 1 (введение, в части описания концепции цифрового двойника и микросервисных систем, стр. 511–512), А.Н. Черных принадлежит раздел 3 (обзор работ в части описания архитектур обработки данных интернета вещей, стр. 516), Х.Л. Гонсалес-Компеану принадлежит раздел 1 (введение, в части описания туманных вычислений и систем поточной обработки данных, стр. 512–513), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 513–525). В работе [3] Г.И. Радченко принадлежит введение (стр. 82–83), А.Н. Черных принадлежит раздел 1.2 (обзор облачных и туманных вычислений стр. 84), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 84–98). В работе [4] А.Н. Черных принадлежит введение (стр. 48–49), Г.И. Радченко принадлежит раздел 1 (обзор технологий виртуализации, стр. 49–51), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 51–79). В работе [5] Г.И. Радченко принадлежит раздел 1 (введение, в части описания концепции цифрового двойника и микросервисных систем, стр. 66), А.Н. Черных принадлежит раздел 3 (обзор литературы в части описания архитектур обработки данных интернета вещей, стр. 71), Х.Л. Гонсалес-Компеану принадлежит раздел 1 (введение, в части описания туманных вычислений и систем поточной обработки данных, стр. 67), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 68–70, 72–80). В работе [6] Г.И. Радченко принадлежит введение, разделы 2-3 (описание концепции цифрового двойника и облачной платформы для поддержки цифровых двойников, стр. 100–102, 103–105), А.Н. Черных принадлежит раздел 1 (обзор смежных работ, стр. 102–103), А.Б.А. Алаасаму принадлежат все остальные

результаты и разделы (стр. 105–116). В работе [7] Г.И. Радченко принадлежат разделы 1, 3, 4 (введение, описание концепции цифрового двойника, описание облачной платформы цифровых двойников, стр. 83–85), А.Н. Черных принадлежит раздел 2 (обзор близких по тематике работ, стр. 84), А.Б.А. Алаасаму принадлежат разделы 5-8 (описание подхода микро-потоков работ, реализация и развертывание экспериментального исследования, оценка производительности микро-потоков работ, заключение, стр. 86–88). В работе [9] Г.И. Радченко принадлежит введение (стр. 0804), А.Н. Черных принадлежит раздел 2 (обзор близких по тематике работ стр. 0805), А.Б.А. Алаасаму принадлежат все остальные результаты и разделы (стр. 0805–0809). В работе [10] Г.И. Радченко принадлежит введение и часть обзора литературы, посвященная описанию общей концепции цифрового двойника (стр. 489), А.Н. Черных принадлежит заключение (стр. 493), К.В. Бородулину принадлежит часть обзора литературы, посвященная поточной обработке данных (стр. 490), А.А. Подкорытову принадлежит часть обзора литературы, посвященная системам обработки потоков работ (стр. 490), А.Б.А. Алаасаму принадлежат разделы описания концепции микро-потоков работ, реализация и тестирование предложенного подхода (стр. 490–493).

Структура и объем работы

Диссертация состоит из введения, четырех глав, заключения и библиографии. В приложении 1 приведены основные аббревиатуры, используемые в диссертации. В приложении 2 приведены основные обозначения, используемые в диссертации. Объем диссертации составляет 147 страниц, объем библиографии – 150 наименований.

Содержание работы

Во введении приводится обоснование актуальности темы и степень ее разработанности; формулируются цели и задачи исследования; раскрываются

новизна, теоретическая и практическая значимость полученных результатов; формулируется методологическая основа диссертационного исследования; дается обзор содержания диссертации.

В первой главе, «Обработка потоков данных в туманных вычислительных средах», рассматриваются понятия модели туманных вычислений и обработки потоков данных в туманных вычислительных средах в контексте систем индустриального интернета вещей и цифровых двойников. Обсуждаются ключевые подходы к организации архитектуры программных систем и методы обработки потоков данных с учетом и без учета состояния, инструменты потоковой обработки данных, а также особенности применения платформ управления научными потоками работ для решения таких задач. Особое внимание уделено обзору методов декомпозиции потоков работ и подходов к проектированию программных систем, ориентированных на обработку потоков данных в туманных средах.

Во второй главе, «Микро-потоки работ», представлена математическая модель микро-потоков работ для обработки потоков данных в распределенных вычислительных средах, таких как туманные вычислительные системы. Модель включает в себя алгоритм рефакторинга зависимостей в монолитном потоке в набор автономных микро-потоков работ. Такое разделение поддерживает независимость реализации, исполнения, разработки, сопровождения и кроссплатформенного развертывания микро-потоков работ на независимых вычислительных узлах.

В третьей главе, «Программная поддержка модели микро-потоков работ», представлены ключевые аспекты реализации и функционирования разработанных программных компонентов, поддерживающих реализацию модели микро-потоков работ. На языке Java был разработан комплекс вычислительных акторов для платформы управления потоков работ Kepler. Разработанные акторы обеспечивают поддержку микро-потоков работ путем реализации ряда типовых операций обработки потоков данных, а также

вершины-потребления и вершины-генератора, обеспечивающих взаимодействие с платформой обработки потоков данных Apache Kafka. С использованием данных акторов, на базе платформы Kepler был разработан набор потоков работ, решающих типовую задачу обработки потоков данных интернета вещей. Для проведения вычислительных экспериментов, также был разработан ряд программных утилит, поддерживающих симуляцию IoT потоков работ, репликацию данных между географически-распределенными локациями серверов Apache Kafka, а также рефакторинг монолитных потоков работ.

В четвертой главе, «Вычислительные эксперименты», представлены результаты реализации и тестирования модели микро-потока работ. Вычислительные эксперименты включают в себя реализацию и тестирование алгоритма по рефакторингу монолитного потока работ на микро-потоки работ; реализацию обработки данных с использованием локальных и удаленных вычислительных ресурсов; сопоставлению применения монолитных и микро-потоков работ для обработки потоков данных; а также оценку возможностей по реализации вычислений с сохранением состояния в контексте экстренной остановки и переноса вычислительного процесса в микро-потоке работ на другой вычислительный узел.

В **заключении** в краткой форме излагаются итоги выполненного диссертационного исследования, представляются отличия диссертационной работы от ранее выполненных родственных работ других авторов, даются рекомендации по использованию полученных результатов и рассматриваются перспективы дальнейшего развития темы.

ГЛАВА 1. ОБРАБОТКА ПОТОКОВ ДАННЫХ В ТУМАННЫХ ВЫЧИСЛИТЕЛЬНЫХ СРЕДАХ

В главе 1 рассматриваются понятия модели туманных вычислений и обработки потоков данных в туманных вычислительных средах в контексте систем индустриального интернета вещей и цифровых двойников. Обсуждаются ключевые подходы к организации архитектуры программных систем и методы обработки потоков данных с учетом и без учета состояния, инструменты потоковой обработки данных, а также особенности применения платформ управления научными потоками работ для решения таких задач. Особое внимание уделено обзору методов декомпозиции потоков работ и подходов к проектированию программных систем, ориентированных на обработку потоков данных в туманных средах.

1.1. Концепция туманных вычислений

В данном разделе рассматриваются основные понятия, связанные с концепцией туманных вычислений.

1.1.1. Облачные вычисления

В основе концепции туманных вычислений лежит понятие облачных вычислений. Впервые идея организации повсеместного удаленного доступа к вычислительным ресурсам зародилась в 1960–70-х годах под названием *коммунальных вычислений (utility computing)* [28,31]. Суть подхода коммунальных вычислений сводится к обеспечению удаленного предоставления пользователям вычислительных услуг, в соответствии с концепцией «оплата по мере использования» так же, как потребители оплачивают потребленное электричество и/или воду. К началу 1990-х годов работы в этой области привели к появлению подхода так называемых *мета-вычислений (metacomputing)*, основанной на идее организации гетерогенных вычислительных ресурсов, связанных специальным промежуточным программным обеспечением таким

образом, что пользоваться ими так же просто, как персональным компьютером [97,126]. Реализация этой концепции базировалась на разрабатываемых в то время системах управления распределенными вычислительными задачами, такими как *Condor* [61,73].

В середине 1990-х годов развитие этого подхода привело к появлению концепции *грид-вычислений (grid computing)*, которая обеспечивает возможность совместного использования ресурсов географически-распределенных вычислительных центров для решения крупных, высоко-масштабируемых вычислительных задач [97]. До 2010 года, в области грид-вычислений было разработано множество решений, обеспечивающих управление вычислительными задачами на базе географически-распределенных гетерогенных вычислительных ресурсов. Среди наиболее популярных решений для организации грид-вычислений можно отметить такие системы как *Globus* [30] и *UNICORE* [99]. С другой стороны, невозможно не отметить набравший в то же самое время популярность подход *добровольных вычислений (volunteer computing)*. В отличие от грид-вычислений, которые были ориентированы на объединение центров обработки данных, высокопроизводительных и суперкомпьютерных систем, ресурсами добровольных вычислений служили простаивающие персональные компьютеры конечных пользователей. Наиболее успешным решением в этой области является платформа *BOINC* [12], на базе которой было реализовано более трех десятков проектов по высоко масштабируемым вычислениям в таких областях, как астрономия, вычислительная химия, биология, астрофизика и др.

Благодаря широкому распространению Интернета и использованию Веба в коммерческих целях, в 2000-х годах фокус проектов управления распределенными вычислительными системами постепенно переходит от предоставления вычислительных ресурсов для специальных целей, на более потребительскую концепцию так называемых *облачных вычислений (cloud computing)* в рамках которой большому количеству независимых конечных

пользователей обеспечивается предоставление набора абстрактных вычислительных ресурсов, по принципу оплаты по мере использования. Облака представляют большой пул виртуализированных ресурсов (таких как аппаратное обеспечение, платформы разработки и/или сервисы), которые могут быть динамически переконфигурированы для адаптации к переменной нагрузке, позволяя также оптимально использовать ресурсы, и обычно используются по модели оплаты по мере использования [110].

1.1.2. Концепция интернета вещей

Экспоненциальное развитие облачных вычислений обеспечило новые возможности обработки данных, поступающих от интеллектуальных датчиков, и привело к стремительному росту внедрения решений в области так называемого *Интернета вещей (Internet of Things, IoT)*. Термин "Интернет вещей" был впервые введен в 1999 году [55]. По данным компании Allied Market Research, мировой рынок датчиков оценивался в $138 965 млн в 2017 году и по прогнозам достигнет $287 002 млн к 2025 году, увеличиваясь с 2018 по 2025 год на 9,5% [127]. Специалисты корпорации Ericsson подсчитали, что по состоянию на 2021 год насчитывается более 28 миллиардов подключенных к Интернету устройств [26]. В то время как по оценкам Бюро переписи населения США в ноябре 2019 года численность населения мира приближается к 7,6 млрд человек [109]. Это означает, что сегодня на каждых двух человек приходится почти семь подключенных устройств. Это можно назвать революцией IoT.

IoT можно определить как взаимосвязь сенсорных устройств и актуаторов, которая обеспечивает возможность обмена информацией между платформами через единую инфраструктуру, что позволяет сформировать единую операционную модель для поддержки инновационных приложений. Это обеспечивается благодаря повсеместному сбору информации, анализу данных и представлению информации с использованием облачных вычислений

как унифицирующей инфраструктуры [40]. Можно выделить следующие ключевые компоненты архитектуры IoT [43,81]:

- *конечные IoT устройства* (датчики, актуаторы), обеспечивающие связь с физическим миром, путем считывания информации о состоянии среды, либо генерацией какого-либо воздействия;
- *системы локальной коммуникации*, включающие в себя системы передачи данных и локальные вычислительные сети, обеспечивающие передачу данных от конечных IoT устройств до ближайших устройств-аггрегаторов;
- *аггрегаторы, маршрутизаторы и шлюзы IoT* – аппаратные компоненты, обеспечивающие сбор, аггрегацию, кеширование, предобработку и трансляцию потоков данных от конечных IoT устройтв в глобальную сеть, а также маршрутизацию обратных потоков управления;
- *глобальная вычислительная сеть*, обеспечивающая возможность обеспечения повсеместного доступа к данным и управления IoT системами из любой географической локации;
- *облачная вычислительная инфраструктура* предоставляет потенциально безграничные вычислительные ресурсы для поддержки приложений аггрегации и анализа данных IoT;
- *системы обработки потоков данных,* обеспечивающие обработку информации, поступающей от устройств IoT в режиме, близком к реальному времени;
- *приложения IoT*, которые предоставляют конечным пользователям интерфейс к возможностям и ресурсам систем IoT. Они могут быть реализованы в виде классических настольных приложений, веб-приложений или приложений для мобильных платформ.

1.1.3. Цифровые двойники

Из-за широкого распространения использования технологий IoT, в настоящее время физический и цифровой миры стали взаимосвязанными, что послужило мотивом для установления взаимосвязи между этими двумя мирами посредством телеметрии, поддерживаемой моделированием. Например, в автогонках Формулы-1 поток данных, собранный с сотен датчиков, установленных на автомобиле, и передаваемый на пульт технического обслуживания, служит источником данных для моделирования работы автомобиля в реальном времени [71]. Используя эти модели, инженеры могут вносить корректировки в режим работы автомобиля удаленно, непосредственно в режиме гонки. Использование таких подходов в индустриальной сфере называется *индустриальным интернетом вещей (Industrial Internet of Things, IIoT)* целью которого является создание *умной индустрии (Smart Industry)* или *Индустрии 4.0 (Industry 4.0)*, которая интегрирует IoT с производственными технологиями для создания взаимосвязанного производственного предприятия, которое анализирует информацию для осуществления интеллектуальных действий в физическом мире [79]. Важным приложением умной индустрии является так называемый *цифровой двойник (Digital Twin, DT)*.

Принято считать, что концепция DT была впервые изложена в начале 2000-х годов в контексте проектирования систем *управления жизненным циклом продуктов (Product Lifecycle Management, PLM)* [38]. DT представляет собой систему, состоящую из трех основных компонентов: физический объект в реальном мире, его виртуальное представление в виртуальном мире, а также потоки данных и управления, которые объединяют реальные и виртуальные компоненты (см. рис. 1) [34].

DT – это интегрированная мульти-физическая, мульти-масштабная вероятностная симуляция сложного изделия, которая использует наиболее

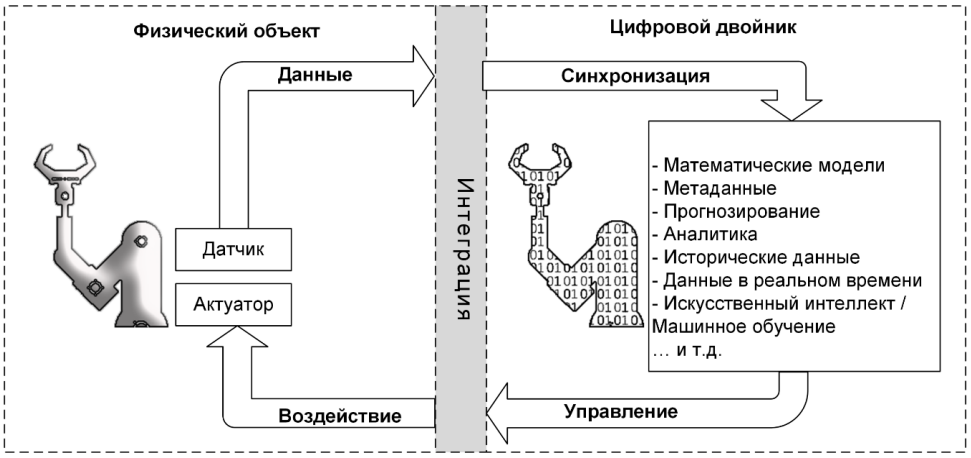


Рис. 1. Концепция цифрового двойника.

подходящие физические модели, актуальные данные сенсоров и др. для того, чтобы получить как можно более достоверное представление соответствующего реального объекта. В отличие от традиционного моделирования, виртуальное представление в DT постоянно обновляется с учетом состояния обслуживания и производительности на протяжении всего жизненного цикла физического объекта [65].

Для создания DT технологических процессов и систем применяются системы математических моделей и методов, таких как интеллектуальный анализ данных, метод конечных элементов и т.д. [58]. Каждый из таких методов предъявляет особые требования к необходимым вычислительным ресурсам. Например, методы интеллектуального анализа данных требуют вычислительных ресурсов, предоставляющих существенные объемы хранения данных [118], в то время как модели, использующие метод конечных элементов, требуют высокопроизводительных вычислительных систем (или суперкомпьютеров) [114].

С одной стороны, предоставление вычислительных ресурсов с требуемыми характеристиками и возможность их динамического масштабирования, предоставляемая облачными вычислительными системами, позволяют сделать вывод, что концепция облачных вычислений удовлетворяет таким требованиям к вычислительной инфраструктуре [85]. С другой стороны, отличительной характеристикой DT является возможность сбора, передачи и

анализа потоков данных от систем IIoT в режимах, близких к реальному времени, для настройки и актуализации их виртуального состояния [20]. В этом случае применение облачных вычислений не позволит удовлетворить требования приложений, чувствительных к времени задержки и местоположению сервисов обработки данных [64]. Возможным решением этой проблемы может служить применение модели *туманных вычислений*, которая расширяет концепцию облачных вычислений, предоставляя вычислительные ресурсы ближе к источникам данных [48].

1.1.4. Туманные вычисления

Одним из наиболее ярких примеров продуктов умной индустрии является *автономный автомобиль (Autonomous vehicle – AV)*. Ожидается, что мировой рынок AV достигнет 52,4 млрд. долларов США в 2027 году при ежегодном росте в 14,5% в течение прогнозного периода с 2021 по 2027 год [128]. В то время как Twitter с 270 миллионами пользователей в 2018 году генерировал около 100 ГБ данных в день, было подсчитано, что один AV может производить до 30 терабайт данных за один день движения [129]. Без эффективного способа управления всеми этими данными, AV столкнется с проблемами пропускной способности и задержки при обработке данных.

Технология облачных вычислений обеспечивает предоставление динамически-масштабируемого, потенциально неограниченного объема вычислительных ресурсов [120]. Но высокая величина задержки при передаче данных *(латентность)* и возможные перегрузки сетевых каналов не позволяют облачным решениям обеспечить полное соответствие требованиям таких систем как DT, чувствительным к временным задержкам и местоположению узлов обработки данных [64]. Удаленное географическое расположение облачных центров обработки данных приводит к неизбежным проблемам с латентностью при передаче и обработке данных, а также существенным ограничениям в пропускной способности каналов связи, расположенных на *краю*

сети (network edge), то есть в непосредственной близости к источникам данных и конечным пользователям. Разработчики концепции *туманных вычислений (fog computing)* пытаются решить эту проблему, перемещая часть задач по обработке и хранению данных из облака ближе к краю сети, в так называемые *туманные узлы (fog nodes)* [19].

Туманные вычисления — это многоуровневая модель, обеспечивающая повсеместный доступ к общему континууму масштабируемых вычислительных ресурсов и поддерживающая развертывание распределенных приложений и сервисов с учетом латентности. Туманная вычислительная среда состоит из туманных узлов (физических или виртуальных), расположенных между интеллектуальными конечными устройствами и централизованными (облачными) сервисами [49]. Туманные вычисления позволяют решить задачу минимизации времени запроса-ответа от/до поддерживаемых приложений и обеспечивают конечным устройствам локальные вычислительные ресурсы и, при необходимости, сетевое подключение к централизованным сервисам (см. рис. 2).

Выделяют следующие ключевые характеристики туманных вычислительных систем [49]:

– *Контекстуальная осведомленность о местоположении и минимизация латентности.* Туманные вычисления решают задачу минимизации латентности при обработке данных благодаря осведомленности тумаными узлами о своем логическом местоположени в контексте всей системы и затрат на связь с другими узлами. В отличие от более централизованного подхода облачных вычислений, сервисы и приложения, на которые нацелены туманные вычисления, уделяют существенное внимание географическому и сетевому расположению вычислительных ресурсов при планировании нагрузки.

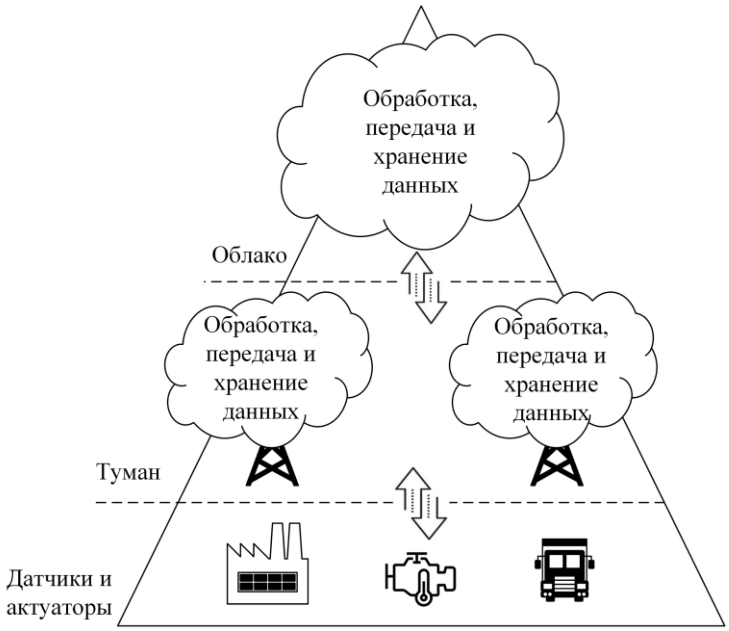


Рис. 2. Общая архитектура туманных вычислений.

– *Гетерогенность.* Туманные вычисления поддерживают обработку данных различных форматов, сбор и передача которых обеспечивается посредством различных сетевых коммуникационных решений.
– *Интероперабельность и федерация.* Бесперебойная поддержка таких сервисов как обработка потоков данных в реальном времени, требуют совместной работы кооперации большого числа компонентов. Компоненты туманной вычислительной среды быть способны к взаимодействию, а сервисы должны распределены между зонами доступности.
– *Масштабируемость и гибкость кластеров туманных узлов.* Туманные вычисления должны обеспечивать адаптивность к требуемым ресурсам на уровне кластеров туманных узлов, поддерживая эластичное предоставление требуемых вычислительных ресурсов, учитывая изменения вычислительной нагрузки и изменения состояния сети.

– *Взаимодействие в реальном времени.* Приложения туманных вычислений предполагают взаимодействие в реальном времени, обработку данных в поточном, а не в пакетном режиме.

Рассмотренные характеристики туманных вычислительных систем подразумевают, что ключевой класс туманных приложений – это географически-распределенные сервисы, обеспечивающие обработку потоков данных, поддерживающие возможность работы в режимах, близких к реальному времени и размещение в непосредственной близости к источникам данных для минимизации латентности. Проектирование, развертывание и поддержка работы таких приложений требует особого подхода от разработчиков программных систем. Ключевым подходом к решению этой задачи может служить применение событийно-управляемой архитектуры, реализуемой с использованием так называемого микросервисного подхода проектирования приложений.

1.2. Событийно-управляемая архитектура и микросервисный подход

1.2.1. Событийно-управляемая архитектура

Туманная вычислительная инфраструктура и приложения IoT ориентированы на обработку потоков данных из различных источников. *Событийно-управляемая архитектура (Event-Driven Architecture – EDA)* наиболее адаптирована к этому типу приложений. EDA — это системная архитектура, состоящая из слабосвязанных, компонентов обработки событий, которые принимают и обрабатывают события одновременно [68]. В рамках данной архитектуры, под *событием* подразумевается некоторая значимая информация о состоянии процесса либо явления, происходящего внутри или снаружи реализуемой системы.

EDA, по своей природе, является экстремально слабосвязанной и высоко распределенной архитектурой программных систем [37]. В контексте текущего исследования, *слабосвязанной (loosely coupled)* архитектурой будем называть такую архитектуру программных систем, при которой компоненты, составляющие данную систему, взаимодействуют друг с другом исключительно на основе открытых интерфейсов, не зависимо от отдельных аспектов внутренней реализации. Такой подход обеспечивает возможность внесения изменений в реализацию компонента без влияния на компоненты, которые его используют, в случае если поведение и ограничения, описанные в его интерфейсе, остаются неизменным [117]. Противоположностью данного подхода является так называемая *сильносвязанная (tightly coupled)* архитектура, в рамках которой взаимодействие между компонентами реализуется с учетом особенности их внутренней реализации, и изменения в реализации одного из них может существенным образом повлиять на реализацию множества других компонентов, составляющих систему.

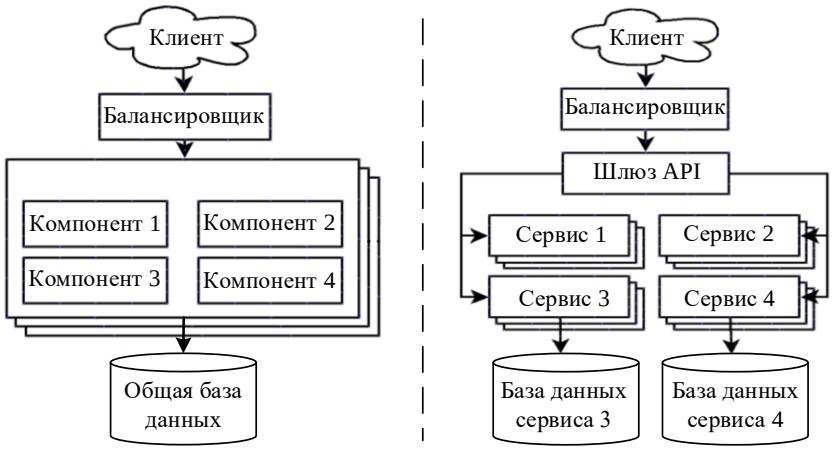
В основе EDA лежат процессы генерации событий источниками данных; коммуникационные каналы и механизмы распространения информации о событиях между обработчиками событий; а также реализация непосредственно обработчиков событий, которые объединяют данные от множества источников, идентифицируют характерные особенности данных и формируют события, содержащие результаты обработки данных [22]. От программных систем, построенных на базе такой архитектуры, ожидается возможность динамического масштабирования предоставляемых вычислительных ресурсов, автоматизации развертывания и управления временем жизни сервисов обработки данных. Иначе развертывание и поддержка подобных систем в масштабах, требуемых для решения реальных задач, становится чрезвычайно сложной либо полностью невозможной. Сегодня предполагается, что такая сложная система должна состоять из мелкоструктурных компонентов [33].

1.2.2. Концепция микросервисов

Концепция микросервисов считается сегодня наиболее перспективной для проектирования слабосвязанных систем, ориентированных на обработку событий. *Микросервисы* – это подход к проектированию распределенных вычислительных систем, в котором приложение декомпозируется на набор независимых сервисов, которые могут быть развернуты, реплицированы и остановлены независимо друг от друга [75,104]. Каждый из таких сервисов ориентирован на реализацию конкретной задачи в рамках предметной области соответствующего приложения.

Особенности данного подхода могут быть ярко проиллюстрированы при его сопоставлении с классическим подходом к проектированию приложений, который также называют *монолитным* (см. рис. 3a) [91]. При проектировании приложения в рамках монолитного подхода, компоненты, отвечающие за реализацию различных областей бизнес-логики, объединены в единый мета-компонент (сервер), и сильно связаны между собой как непосредственными вызовами методов других компонентов, так и косвенно, посредством чтения/записи данных в базу данных, общую для всех компонентов данного приложения. В отличие от монолитного подхода, выделяются следующие особенности реализации систем в соответствии с концепцией микросервисов (см. рис. 3b) [52]:

– микросервисы должны использовать открытые, легковесные механизмы для организации коммуникации (чаще всего базирующиеся на стеке протоколов TCP/IP и HTTP);
– разделение приложения на микросервисы должно происходить на основе выделения и изоляции частей бизнес-процессов;



(a) Монолитная система (b) Микросервисная система

Рис. 3. Сопоставление монолитного подхода и концепции микросервисов [91].

– микросервисный подход должен обеспечить независимое управление жизненным циклом экземпляров каждого компонента, включая возможности независимого развертывания, масштабирования и остановки.

Применение концепции микросервисов позволяет преодолеть существенные недостатки монолитного подхода, включая такие проблемы как распределение вычислительной нагрузки, обеспечение бесперебойного функционирования системы во время обслуживания или обновления ее компонентов. При этом, микросервисный подход позволяет осуществлять независимую разработку, развертывание, масштабирование и миграцию микросервисов с одного вычислительного ресурса на другой [91].

Несмотря на достоинства данного подхода, такая гибкость не бывает бесплатной. Например, коммуникационные затраты увеличиваются из-за необходимости организации обмена данными между микросервисами, что, в свою очередь, повышает сложность управления потоками данных.

1.3. Обработка потоков данных

Отличительной особенностью туманных приложений, связанных с платформами интернета вещей, является сбор и обработка информации об объекте

физического мира в режиме, близком к реальному времени. Для решения этой задачи вычислительные сервисы должны уметь обрабатывать потоки данных, формируемые сенсорами, собирающими информацию о состоянии физических объектов [16]. Производительность таких систем критически зависит от способности безопасно и эффективно собирать, передавать и анализировать потоки данных между объектами реального мира и системами обработки данных [96].

Потоком данных (data stream) называют счетную, неограниченную последовательность элементов данных, которые становятся доступными с течением времени, например, показания датчиков IoT или котировки акций в финансовых приложениях [67]. *Потоковой обработкой данных (data stream processing)* называют выполнение различных задач обработки над потоком данных, которые выполняются непосредственно в процессе поступления новых данных (в отличие от методов пакетной обработки данных, которые применяются в случае, когда все необходимые данные уже собраны в отдельном хранилище) [32] (см. рис. 4). Эти задачи могут состоять в построении моделей связанных с прогнозированием значений следующих элементов, обнаружении частых закономерностей или аномальных значений.

Согласно [35], алгоритмы потоковой обработки данных обрабатывают последовательно потенциально неограниченные входные потоки, позволяя получить поток результатов как на основе полного набора ранее обработанных данных, так и на основе ограниченного набора последних данных, отобранных посредством метода скользящего окна.

Подход потоковой обработки данных часто противопоставляется методам *пакетной обработки данных*. При реализации пакетной обработки данных, например, с использованием концепции MapReduce, все входные

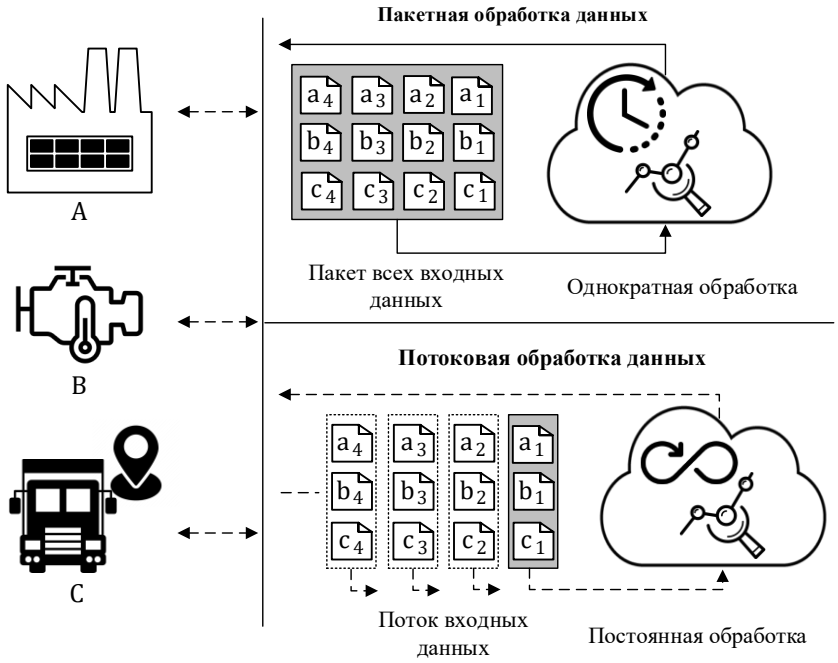


Рис. 4. Архитектурные различия между системами потоковой и пакетной обработки данных.

данные должны быть предварительно собраны перед началом вычислительного процесса. Одним из наиболее популярных решений для поддержки концепции MapReduce является Hadoop. *Hadoop* — это программная платформа для приложений, обеспечивающих надежную и отказоустойчивую обработку больших данных на базе кластерных вычислительных систем (вплоть до тысяч узлов) [130].

В процессе реализации алгоритма MapReduce, входной набор данных разбивается на большое число независимых фрагментов, которые могут быть параллельно обработаны на фазе map. Выходные результаты такой обработки в последствии поступают на вход задач reduce. Платформа Hadoop заботится о планировании заданий, их мониторинге и повторном выполнении неудачных заданий. При этом, накладные расходы, связанные с подготовкой вычислительной задачи (такие как планирование, копирование или перенос данных и кода), не позволяют обеспечить оперативную обработку данных, поступающих в режиме близком к реальному времени [94]. Варианты использования этих концепций также отличаются. Можно привести

следующие примеры задач, для решения которых используется пакетная обработка данных:

– система расчета заработной платы сотрудников, которая собирает все соответствующие данные и обрабатывают их в пакетном режиме в конце определенного периода, например, каждого месяца;
– система виртуальных краш-тестов, которая проводит моделирование на основе предварительно определенной и параметризованной модели автомобиля и препятствия.

Примерами задач, для решения которых применяется потоковая обработка данных, являются:

– система обнаружения мошеннических транзакций в онлайн-режиме для идентификации и предотвращения аномальных транзакций в режиме реального времени;
– мониторинг транзакций фондового рынка;
– мониторинг потоков данных от систем интернета вещей для идентификации состояния и прогнозирования течения технологических процессов.

Традиционно, системы пакетной обработки данных, применяются для решения задач высокопроизводительной обработки больших данных. Решение таких задач может занимать несколько часов и даже дней [3,54].

1.3.1. Компоненты систем обработки потоков данных

Можно выделить следующие элементы систем, обеспечивающих обработку потоков данных [13,89].

– *Сообщение* – это основной элемент данных, обрабатываемый приложением при получении из исходного потока данных. Сообщение состоит из набора атрибутов, каждому из которых присвоено определенное значение.

- *Генератор (продюсер)* – это компонент системы, обеспечивающий постоянную генерацию сообщений на основе оригинального источника данных.
- *Поток данных* – это последовательность сообщений, которые могут быть потребелны и обработаны потребителем.
- *Потребитель* – это система, отвечающая за получение и обработку сообщений из потока данных.
- *Оператор* – базовый функциональный элемент потокового приложения. Оператор получает данные из входных потоков, применяет заранее определенную функцию над входящими сообщениями и генерирует выходной поток с сообщениями-результатами обработки.
- *Схема сообщений* – это спецификация, определяющая общую структуру сообщений в системе и их атрибуты. Каждый отдельный атрибут сообщения может быть рекурсивно определен посредством дочерней схемы. Например, Apache Avro [131] представляет собой платформу для управления схемами сообщений, определенными на языке JSON. Схемы Avro формируются из набора базовых типов (таких как Boolean, long, int, double, float, byte, и string) и структур данных (record, array, enum, map, fixed, и union).

1.3.2. Классификации операций над потоками данных

Можно выделить следующие ключевые классы операторов над потоками данных [13,36,76].

1) *Операторы агрегации*, обеспечивают агрегацию подмножеств входящих сообщений из одного или нескольких потоков данных. Подмножество сообщений, над которым выполняется операция агрегации, называется окном. В зависимости от применяемых правил выделения

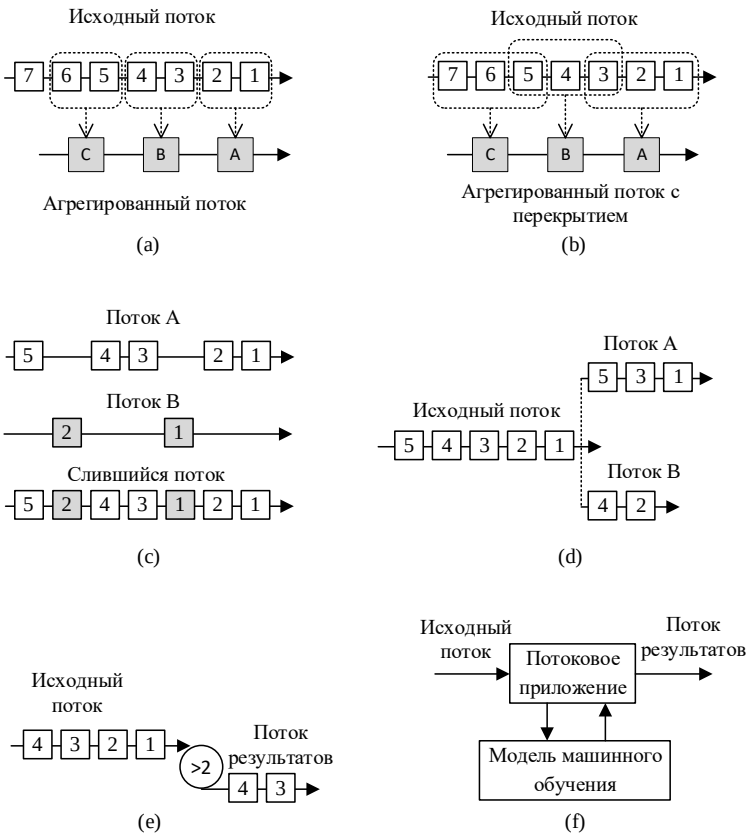


Рис. 5. Примеры базовых операторов над потоками данных.

подмножеств, окна агрегации могут быть различных размеров, непересекающимися (см. рис. 5a) или перекрывающимися (см. рис. 5b). Перекрывающиеся окна подразумевают наличие сообщений, которые попадают одновременно в несколько последовательных окон.

2) *Операторы слияния* обеспечивают объединение нескольких входных потоков данных в один общий поток на основании определенных требований к выравниваю сообщений или других определенных условий (см. рис. 5c).

3) *Операторы расщепления* обеспечивают разделение единого потока данных на несколько потоков в соответствии с заранее определенным алгоритмом (см. рис. 5d).

4) *Логические, математические и специализированные операторы обработки данных*, обеспечивают преобразование входящих сообщений в соответствии с заранее определенными логическими или

математическими правилами (см. рис. 5e). Частным случаем операторов такого типа могут служить операторы, обеспечивающие выполнение методов интеллектуального анализа данных или машинного обучения над входящими потоками данных (см. рис. 5f).

1.3.3. Архитектура систем обработки потоков данных

Можно выделить два ключевых слоя, необходимых для организации системы обработки потоков данных: *слой хранения (storage layer – SL)* и *слой обработки (processing layer – PL)*. SL должен обеспечивать возможность организации высокой пропускной способности для операций ввода/вывода для больших потоков данных. PL отвечает за потребление данных из SL, обработку этих данных и оповещение SL об обновлении данных [7]. Для организации распределенной обработки потоков данных на множестве независимых вычислительных устройств, требуется некоторое расширение представленной модели (см. рис. 6) [50]:

– *Слой потребления потока данных*, включает операции по получению потоков данных из первоначальных источников и передаче в систему обработки или хранения данных [69].
– *Слой обработки потока данных*, это слой, на котором выполняются приложения для обработки потоковых данных. На нем могут размещаться как отдельные приложения обработки данных, так и *платформы обработки потоков данных (Data Stream Processing Engines – DSPE)* [50]. DSPE – это инструмент, включающий операторы обработки потоков данных, которые могут быть настроены и организованы в виде направленного ациклического графа для построения конвейера обработки потока данных [39,44].
– *Слой хранения данных*, обеспечивает хранение сообщений, промежуточных и финальных результатов обработки, выявленных паттернов и информации из потока данных [25]. Различные решения

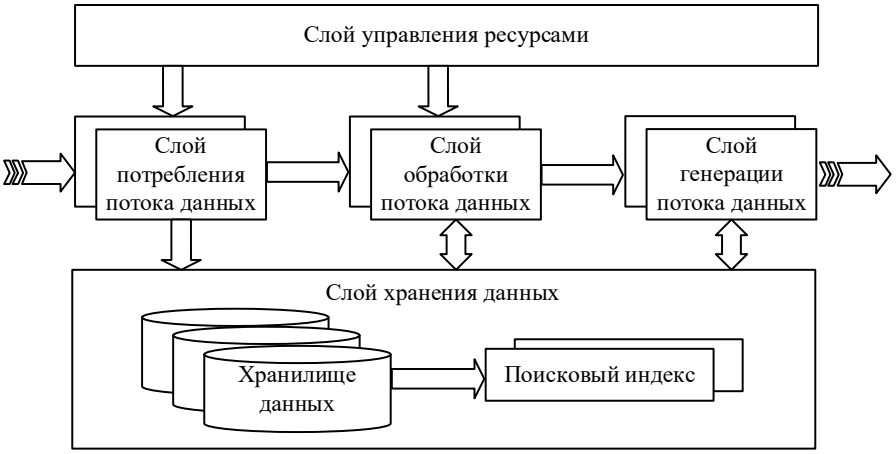


Рис. 6. Архитектура системы обработки потоков данных [50].

в области решения данных могут обеспечить поддержку такого слоя, включая хранилища «ключ-значение», реляционные и NoSQL базы данных, базы данных в памяти [53]. Также, на этом слое могут размещаться решения индексации и полнотекстового поиска в текстовых данных, такие как ElasticSearch или Apache Solr [132].

– *Слой управления ресурсами* отвечает за организацию работы и взаимодействие вычислительных узлов, узлов хранения, а также за управлением временем жизни данных ресурсов (включая выделение новых и освобождение занятых ресурсво) для поддержки обработки больших объемов потоков данных [111]**.**

– *Слой генерации потока данных* отвечает за формирование результирующего потока данных и его отправку последующим потребителям. В качестве таких потребителей могут выступать сторонние приложения, другие системы обработки данных, системы визуализации и мониторинга и др. [25]. Также, извлеченная информация и знания могут быть перенаправлены для врменного или постоянного хранения в хранилище данных для дальнейшего анализа [50].

1.3.4. Обработка потоков данных с сохранением состояния

Процессы, зависящие только от текущего локального состояния, т.е. не от недавнего прошлого или расширенной истории всех таких состояний, называются процессами *без сохранения состояния (stateless)*[1]. Однако разработка сервисов, ориентированных на обработку данных IoT в туманных вычислительных средах, подразумевает необходимость сохранения состояния [78]. Такие системы принято называть *с сохранением состояния (stateful)*[2]. 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 [80]. Разработчики Stateful-систем сталкиваются с рядом проблем, включая ограничения масштабируемости и необходимость дополнительных вычислительных затрат на поддержку управления состоянием [74].

Хранение состояния внутри вычислительного сервиса ограничивает возможности управления его жизненным циклом, поэтому состояние должно храниться в отдельном ресурсе [74], например, внешней базе данных, внешней файловой системе, системе кэширования или в промежуточном программном обеспечении для обмена сообщениями. Тем не менее, отдельный ресурс для работы с состоянием приводит к необходимости выделения дополнительных серверных ресурсов, которые могут включать в себя дополнительные вычислительные мощности, память и хранилища данных. Кроме того, такой подход может приводить к проблемам в масштабируемости. Например, когда экземпляры одного и того же сервиса в одно и то же время обновляют состояние, а другой сервис читает состояние во время

---

[1] В дальнейшем, процессы, сервисы и системы, функционирующие в режиме без сохранения состояния будут называться Stateless.

[2] В дальнейшем, процессы, сервисы и системы, функционирующие в режиме с сохранением состояния будут называться Stateful

обновления, или же читает старое состояние. Такое решение не обеспечит правильность обработки состояния при выполнении операций. Поэтому зачастую состояние должно обрабатываться последовательно. В связи с этим, любая Stateful-операция сталкивается со сложностью управления состоянием. Сложность возрастает в условиях туманной вычислительной среды, где важное значение имеет возможность 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 [133]. В этом случае вычислительный процесс приостанавливается только на время передачи общего состояния, после чего вычисление возобновляется на целевом узле. Реализация механизма, обеспечивающего правильность обработки, хранения и восстановления состояния, является существенным процессом в Stateful-операциях.

1.3.5. Stateful вычислительная инфраструктура

Технологии *виртуализации* позволяют разделить физический вычислительный узел на несколько *виртуальных машин (Virtual Machine – VM)*, каждая из которых имеет собственную изолированную операционную систему и приложения. Они координируются слоем программного обеспечения под названием *гипервизор*, который обеспечивает поддержку параллельного выполнения нескольких VM на одной физической машине [93].

Тем не менее, накладные расходы, связанные с использованием VM в облачных вычислениях, могут ограничить эффективность вычислительных ресурсов. Эта проблема может быть решена при помощи технологии контейнеризации [87]. *Контейнеризация* позволяет запускать независимые

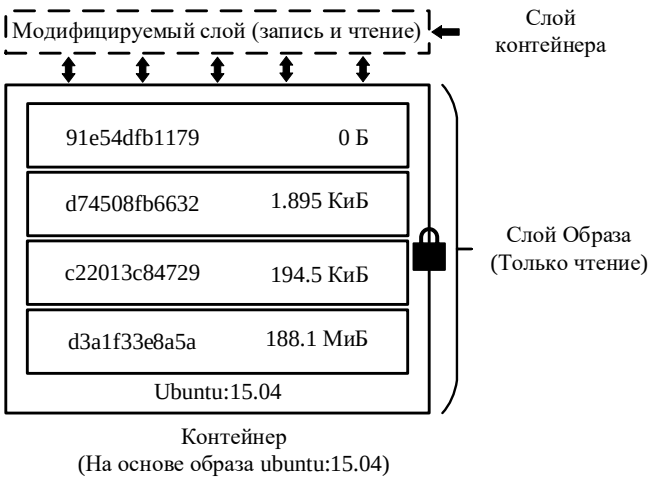


Рис. 7. Многослойная схема организации образов и контейнеров на платформе Docker [135].

контейнеры в виде отдельных процессов непосредственно на ядре базовой операционной системы, обеспечивая легкую изоляцию процессов внутри контейнеров. Одной из наиболее популярных платформ, обеспечивающих контейнеризацию, является Docker. При создании нового контейнера в Docker, новый слой (слой контейнера), доступный для записи, создается поверх слоев, доступных только для чтения (слой образа), а все изменения, внесенные в контейнер, записываются только на слой контейнера (см. рис. 7).

Все базовые образы контейнеров доступны в репозитории образов, например, Docker Hub. Чтобы запустить любой контейнер на новом вычислительном узле, необходимо загрузить только базовый образ из репозитория Docker, после чего из этого образа создается новый уровень контейнера. Сложность проектирования stateful-систем, зависит от возможностей базовой вычислительной инфраструктуры. Stateful вычислительной инфраструктурой будет называться, такая инфраструктура виртуализации, которая позволяет хранить и управлять состоянием внутри контейнера в течение длительного промежутка времени. Технология VM обеспечивает возможность создания моментальных снимков и восстановления состояния приложений. Это позволяет осуществлять миграцию VM между физическими хостами с минимальным влиянием на запущенные сервисы [23]. Такие технологии

применяются во многих платформах управления VM, таких как VMware vCenter [134]. Снимки состояния VM позволяют обеспечить миграцию VM между вычислительными узлами без существенного прерывания вычислительного процесса и воздействия на их состояние. К сожалению, такой подход не распространен в случае применения технологии контейнеризации. Например, реализация такой миграции не так проста при использовании платформы Docker [135].

Методы создания контрольных точек состояния контейнера в настоящее время находятся только в тестовой версии Docker [136]. 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 [137]. Также, CRIU выполняет операцию «PID dance», чтобы восстановить процесс с таким же PID. Эта операция требует привилегированного доступа и обладает низкой производительностью, так как требует множества системных вызовов и может привести к возникновению состояния гонки [88]. С другой стороны, контейнеры Linux LXD [138] поддерживают возможность создания снимков и восстановления контейнеров для резервного копирования или миграции. Но, для целей живой миграции, LXD все еще базируется на CRIU [139].

Между тем, некоторые платформы оркестрации многоконтейнерных систем, такие как Kubernetes, предоставляет контроллер StatefulSet для управления stateful-приложениями [140]. Тем не менее, его работа по-прежнему подлежит некоторым ограничениям, таким как потеря состояния при реализации «rolling-обновлений», сложность или невозможность применения балансировщиков нагрузки и др. [140]. Ключевой же проблемой в управлении StatefulSet является отсутствие механизмов автоматического создания

замены вышедшего из строя StatefulSet [2]. Поэтому решение проблемы сохранения состояния при использовании технологии контейнеризации является активной областью исследований. Это особенно важно при обработке Stateful-потоков данных, где каждая точка данных может играть решающую роль в системе.

1.3.6. 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 [141] 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 [77]. 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. В этом случае необходимо учитывать, что потоки данных генерируются несколькими источниками данных, а также они часто должны обрабатываться последовательно, посредством применения механизмов stateful-обработки временных рядов, таких как скользящее среднее и др. [9].

1.3.7. Платформы обработки потоков данных

В данном разделе рассматриваются наиболее распространенные сегодня платформы, обеспечивающие поддержку обработки потоков данных. Распределенная вычислительная платформа Apache Flink управляет состоянием, создавая локальные постоянные состояния внутри приложения. Для обеспечения отказоустойчивости Flink предоставляет механизм синхронизации мгновенных снимков состояния со внешними ресурсами, такими как HDFS и/или Amazon S3 [142]. Аналитическая платформа для обработки больших объемов данных Apache Spark [143] также обеспечивает функцию stateful обработки данных, где состояние обработки потока данных хранится в локальной памяти, но для отказоустойчивой Spark сохраняет данные в отказоустойчивую систему хранения, такую как HDFS. Apache Kafka это отказоустойчивая очередь сообщений с *программным интерфейсом приложений (Application Programming Interface – API)*. Kafka хранит данные в «темах» (Kafka topics), которые представляют собой названия категорий, в рамках которых публикуются и потребляются сообщения. Kafka предоставляет следующие API [144]:

– *Producer API* для генерации потоков данных в темы Kafka;
– *Consumer API* для потребления потоков данных из тем Kafka;
– *Streams API* для трансформации потока данных из исходной темы Kafka в конечную тему Kafka. Например, Kafka Streams DSL (Domain Specific Language) API, автоматически создает и управляет хранилищами состояния приложений при вызове Stateful-операций. Kafka Streams DSL обеспечивает автоматическое возвращение состояния путем синхронизации локального хранилища состояния с самим промежуточным программным обеспечением Kafka [145].

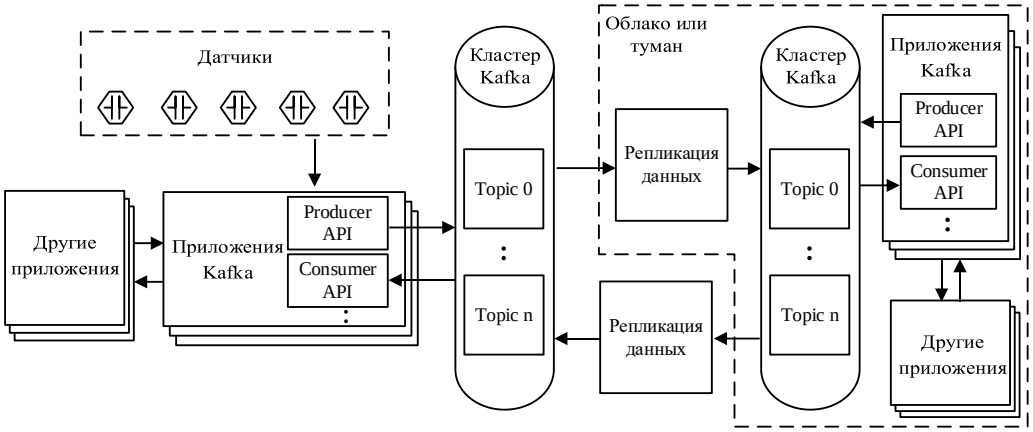


Рис. 8. Общая архитектура распределенной системы на основе платформы Apache Kafka.

– *Connect API* для постоянного потока данных из не-Kafka приложения (база данных, файл и т.д.) в тему Kafka или наоборот;
– *Admin API* для управления компонентами кластера Kafka.

Обмен данными между сервером Kafka и клиентами осуществляется по протоколу TCP. Клиент инициирует соединение, отправляет последовательность сообщений запроса на сервер, после чего получает сообщения ответа. Каждый потребитель потока данных маркируется *идентификатором группы потребителей* (*Group ID*), и каждое сообщение, опубликованное в теме, доставляется одному экземпляру потребителя в рамках каждой Group ID. Kafka также поддерживает семантику обработки сообщений «ровно один раз» (*exactly-once processing semantics*), гарантируя, что каждая запись будет обработана один и только один раз, даже если в процессе обработки произошел какой-то сбой на клиентах или брокерах Kafka [146]. На рис. 8 представлена общая архитектура распределенной системы на основе платформы Apache Kafka.

Можно выделить три основных сценария приложений, взаимодействующих с Kafka Streams API. Во-первых, приложение может служить адаптером для не-Kafka приложений (такие как база данных, файл и т.д.), для обеспечения связи с кластером Kafka. Во-вторых, приложение может обеспечить сбор данных с датчиков. В-третьих, приложение может выступать в качестве

Табл. 1. Сравнение платформ обработки потоков данных: Kafka Stream API, Flink и Spark

Критерий	**Kafka Streams API**	**Flink и Spark**
Развертывание	API, который может быть встроен в приложение и не навязывает конкретный метод развертывания	Фреймворк, развертывающий приложение, либо в автономных кластерах, либо с использованием YARN, Mesos или контейнеров
Работа с состоянием, отказоустойчивость	Состояние хранится локально и синхронизируется с собственной очередью Kafka	Состояние хранится локально, но для обеспечения отказоустойчивости может быть сконфигурировано внешнее хранилище, например HDFS
Источник потоковых данных	Только из очереди сообщений Kafka, которая поддерживает Connect API. Может быть использован Producer и Consumer API для решения проблемы ввода/вывода данных из других систем	Kafka, другие очереди сообщений, файловая система и другие внешние системы
Результат обработки данных	Kafka, состояние приложения, база данных или любая внешняя система	Kafka, другие очереди сообщений, файловая система и другие внешние системы
Ограниченные и неограниченные потоки данных	Неограниченные потоки данных	Ограниченные и неограниченные потоки данных

автономной единицы обработки данных на основе Kafka Streams DSL API. Для репликации данных Kafka в нескольких географических центрах обработки данных, в настоящий момент существует два официальных инструмента MirrorMaker и Confluent Replicator [147].

Каждая из платформ Kafka, Flink, и Spark имеет свои особенности, которые следует учитывать при реализации операций с сохранением состояния. В табл. 1 приведены ключевые архитектурные различия между Kafka Stream API, Apache Flink и Apache Spark [66,148]. Однако системы управления потоками данных обычно не накладывают ограничений на организацию процесса обработки и преобразования, реализуемых компонентами обработки данных. Это может привести к сложностям при проектировании и модернизации сложных конфигураций систем обработки данных.

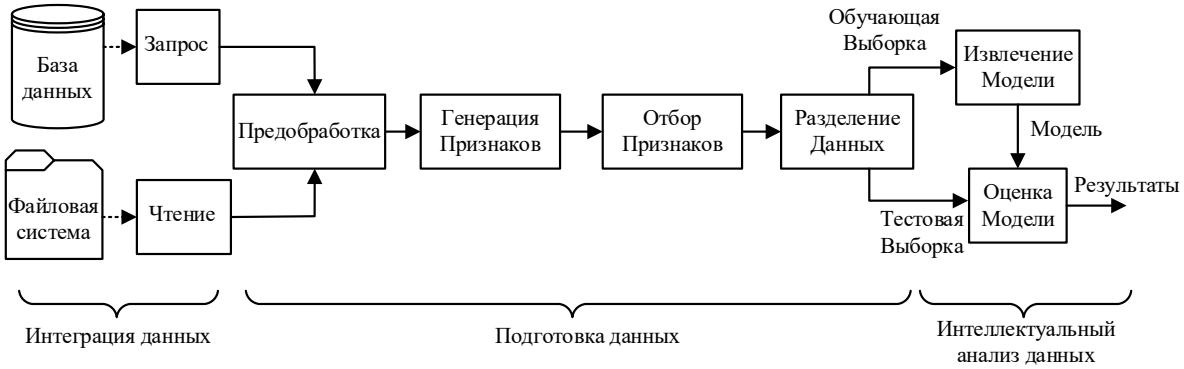


Рис. 9. Пример потока работ в научных экспериментах [59].

Сегодня, наиболее развитой концепцией для решения подобного класса задач является подход представления обработки данных в виде так называемых *потоков работ (workflow)*. Для поддержки подобных решений применяются системы управления научными потоками работ.

1.4. Научные потоки работ

1.4.1. Определение научного потока работ

Процесс извлечения знания из данных, полученных в результате моделирования, наблюдений или экспериментов часто включает в себя следующие типовые этапы (рис. 9) [59]:

1) получение данных из определенных источников;
2) подготовка данных, включая их очистку и нормализацию;
3) анализ данных и построение модели на основе части обработанных данных;
4) проверку подготовленной модели на оставшихся данных;
5) визуализацию результатов и подготовку отчетов по результатам анализа;
6) перемещение результатов анализа в систему хранения.

Такие процессы могут быть смоделированы в виде так называемых *научных потоков работ (Scientific Workflow – SWF)*. *Научным потоком работ*

называют набор взаимосвязанных вычислительных задач и задач по обработке данных, направленных на достижение конкретной цели, в частности на проведение вычислительного эксперимента [59].

Необходимо отметить, что кроме научных потоков работ, выделяют также потоки бизнес-операций (Business Workflow), которые, несмотря на общие подходы, не так сильно завязаны на процессы передачи и обработки больших данных, не ориентированы на реализацию обработки данных в режимах, близких к реальному времени и др. [17].

Будем говорить, что приложение w представляет собой SWF, если это распределенное приложение, состоящее из набора действий A, которые обрабатываются в четко определенном порядке для достижения конкретной цели [84]. В этом случае приложение w может быть определено как пара:

$$w = \left(A, \vec{D}\right), \quad (1)$$

где:

– A – набор действий,

– $\vec{D}$ – набор зависимостей. Каждая зависимость $\vec{d} \in \vec{D}$ может представлять собой либо зависимость по управлению, либо зависимость по данным, связывающую упорядоченную пару вершин $(a_m,\ a_n)$, где $a_m, a_n \in A$. Наличие такой зависимости означает, что задача a_n может быть выполнена только после того, как завершится выполнение задачи a_m.

Действием $a \in A$ в SWF называют вычислительную задачу. Она может представлять собой различные вычислительные объекты, такие как веб-сервисы, исполняемые файлы, и др., которые вызываются в процессе исполнения SWF.

1.4.2. Модели представления потоков работ

Для формального описания научных потоков работ могут применяться различные модели представления. Одной из наиболее распространенных является модель *ориентированного ациклического графа (Directed acyclic graph – DAG)*. В этом случае действия, составляющие поток работ, представляются в виде вершин графа, а зависимости между ними представляются в виде направленных ребер графа [70]. В этом случае научный поток работ может быть формально представлен в виде графа G:

$$G = (V, E), \quad (2)$$

где:

– $V = \{v_o, v_1, \dots, v_m\}$ — множество вершин, представляющих собой множество действий SWF

– $E \subseteq (V \times V)$ — бинарное отношение, представляющее зависимости $(v_i, v_j) \in E$ между действиями SWF. При этом, ацикличность графа обеспечивается требованием, что транзитивное замыкание E^+ отношения E является иррефлексивным, т.е. $(v, v) \notin E^+$ для всех $v \in V$ [29].

Наличие связи между вершинами может интерпретироваться как зависимость по данным (в этом случае, зависимое действие требует данных, полученных в результате выполнения всех предыдущих действий), либо по управлению (в этом случае последующее действие может быть выполнено только после выполнения всех действий, от которых оно зависит).

Особенностью DAG, представляющего собой SWF является наличие специальных вершин v_{entry} и v_{exit}. Вершина v_{entry} представляет собой начальное действие, с которого начинается выполнение SWF. У нее отсутствуют входные ребра. Вершина v_{exit} представляет собой конечное действие, которым заканчивается выполнение SWF. У нее отсутствуют выходные ребра. Если выполнение начинается либо заканчивается одновременно

несколькими действиями, то в G могут быть добавлены виртуальные вершины v_{entry} и v_{exit}, а также виртуальные ребра, соединяющие их с соответствующими начальными и конечными [90].

Существует ряд подходов к представлению графов, каждый из которых обладает своими достоинствами и недостатками [42]. Одним из наиболее распространенных подходов является использование модели *матрицы смежности* (*adjacency matrix*) [125]. Матрица смежности графа $G = (V, E)$, состоящего из n вершин определяется как матрица M размерностью $n \times n$, в которой значение $M_{i,j}$ обозначает наличие ребра между вершинами v_i и v_j, где:

$$m_{i,j} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0, & (v_i, v_j) \notin E \end{cases} \quad (3)$$

Еще одной распространенной моделью формализации SWF является модель *сетей Петри (Petri net)*. Классическая сеть Петри представляет собой ориентированный двудольный граф, состоящий из двух типов вершин: *позиций (places)* и *переходов (transitions)*. В графической нотации позиции представляются кружками, переходы обозначаются прямоугольниками. Вершины соединяются направленными ребрами. Соединение вершин одного типа запрещено [115]. Согласно [115,116] сеть Петри PN может быть определена как тройка:

$$PN = (P, T, A), \quad (4)$$

где:

– $P = \{p_o, p_1, \dots, p_m\}$ – конечное множество позиций;

– $T = \{t_0, t_1, \dots, t_n\}$ – конечное множество переходов, $P \cap T = \emptyset$;

– $A \subseteq (P \times T) \cup (T \times P)$ – множество ребер (отношение потока).

Необходимо отметить, что в рамках данного определения авторы ограничиваются ребрами веса 1, так как в контексте моделирования потоков работ позиции соответствуют условиям, а переходы – задачам.

Позиция p называется *входной позицией* перехода t, если существует направленное ребро из p в t. Позиция p называется *выходной позицией* перехода t, если существует направленное ребро из t в p. Аналогичным образом определяются входные и выходные переходы для заданной позиции. Для обозначения множества входных вершин для вершины $x \in P \cup T$ используется обозначение ${}^\bullet x$:

$${}^\bullet x = \{y \in P \cup T : (y, x) \in A\} \quad (5)$$

Аналогичным образом, множество выходных вершин для вершины x, обозначается $x^\bullet$:

$$x^\bullet = \{y \in P \cup T : (x, y) \in A\} \quad (6)$$

В любой момент времени каждая позиция может содержать ноль либо больше *токенов,* в графической нотации отмечаемых черными точками. Наличие токена в определенной позиции означает, что выполнены условия, определяемые данной позицией. *Состояние* сети, также обозначаемое как *маркировка (marking)*, представляет собой распределение токенов по позициям, т.е. функцию $M : P \rightarrow \mathbb{N} = \{0, 1, 2, \dots\}$. Таким образом, маркировка M определяет вектор $m = \{M(p_1),\ M(p_2), \dots,\ M(p_m)\}$, где каждый элемент обозначает количество токенов, находящихся в соответствующей позиции.

Переход PN из одного состояния в другое определяется наличием *активных переходов.* Переход $t \in T$ является активным, если все его входные позиции содержат хотя бы по одному токену: $\forall p \in {}^\bullet t : M(p) > 0$. *Запуск* активного перехода t приводит к *потреблению* одного токена из каждой входной позиции, входящей в ${}^\bullet t$ и генерации одного токена на каждой выходной позиции, входящей в $t^\bullet$.

Если для пары $n_1, n_k \in P \cup T$ есть последовательность вершин $(n_1, n_2\ \dots, n_k)$, что $1 \leq i \leq k-1 : (n_i, n_{i+1}) \in A$, такую последовательность вершин называют *путь,* а также говорят, что для пары n_1, n_k истинно

отношение *достижимости* $R(n_1, n_k)$ [109]. Сеть PN называют *сильно связанной*, если для каждой пары вершин $x, y \in P \cup T$, $R(x, y) = true$.

Моделирование потоков работ посредством сетей Петри реализуется следующим образом: *действия* потока работ моделируются *переходами*; *условия* запуска тех или иных действий потока работ (то есть зависимости между действиями) моделируются *позициями*, а *экземпляр* выполняемого потока работ моделируется набором токенов.

Для моделирования потоков работ часто применяют специальный класс сетей Петри, которые называют *WF-net (WorkFlow net)*. PN называют *WF-net* тогда и только тогда, когда:

1) В PN выделены две специальные позиции: i и o. Позиция i – это исток, т.е. ${}^\bullet i = \emptyset$. Позиция o – это сток, т.е. $o^\bullet = \emptyset$.
2) Если в PN добавить переход t_*, соединяющий i и o, т.е. ${}^\bullet t_* = \{o\},\ t_*{}^\bullet = \{i\}$, то получившаяся сеть Петри будет сильно связанной.

1.4.3. Система управления научными потоками работ Kepler

Приложения в виде SWF сегодня применяются для поддержки сложных вычислительных задач, требующих эффективного управления большими данными, которые обычно хранятся и обрабатываются на базе гетерогенных распределенных вычислительных ресурсов [95]. *Система управления научными потоками работ (Scientific Workflow Management System – SWfMS)* – это программная система, которая обеспечивает среду выполнения SWF, поддерживая задачи определения, создания и выполнения научных потоков работ [62]. SWfMS доказали свою эффективность при реализации научных вычислений, обеспечивая спецификацию потоков работ, координацию вычислительных процессов, планирование и выполнение SWF, отслеживание происхождения данных, поддержку отказоустойчивости и др. [121]. В качестве примеров SWfMS, можно привести такие решения как DiVTB [86], Pegasus [24], Kepler [11], ASKALON [27], и tGSF [45,46].

Платформа Kepler является одной из наиболее популярных SWfMS. Основной целью Kepler является поддержка различных сценариев выполнения вычислительных экспериментов, где пользователь может разрабатывать и использовать его модули для управления выполнением SWF в различных средах, включая частные вычислительные ресурсы [82]. Лежащий в основе Kepler механизм управления потоками работ обеспечивает проверку происхождения данных и воспроизводимости кода, выполняет оркестрацию потоков данных и автоматизирует выполнение потоков работ на базе гетерогенных вычислительных ресурсов [119].

Kepler предоставляет графический интерфейс для построения потоков работ. В Kepler отдельные действия в потоке работ называются *акторами (actors)*. Каждый актор предназначен для выполнения конкретной независимой задачи, которая может быть реализована как атомарная (atomic) или составная (composite) [63]. *Составные акторы (или подпотоки, sub-workflows)* состоят из атомарных акторов, объединенных вместе для выполнения сложных операций. Акторы в потоке работ могут содержать *порты*. Порты обеспечивают потребление и генерацию наборов данных, называемых *токенами,* а также реализуют возможности взаимодействия с другими акторами в потоке работ по каналам связи [113].

Параметры выполнения потока работ задаются независимой сущностью, называемой в Kepler *директором*. Директор определяет, как выполняются акторы и как они общаются друг с другом. Модель выполнения, определяемая директором, называется *моделью вычислений* [63]. Поскольку директор отделен от структуры потока работ, пользователь может изменить модель вычислений, заменив директора с помощью графического интерфейса пользователя Kepler. В результате поток работ может выполняться либо последовательно, например, с помощью директора *синхронного потока данных*

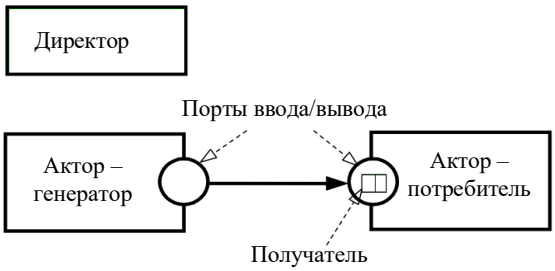


Рис. 10. Семантика взаимодействия компонентов
в платформе Kepler [63].

(Synchronous Data Flow – SDF), либо параллельно, например, с помощью директора *сети процессов (Process Network –PN)* [113] (см. рис. 10). Пользователь может создавать собственных директоров и акторов, а также модифицировать или использовать существующих директоров и акторов.

Тем не менее, сегодня можно выделить несколько ключевых проблем, связанных с использованием SWfMS. Во-первых, платформы SWfMS не ориентированы на обработку потоков данных [15]. SWF исторически ориентированы на исполнение вычислительных задач в пакетном виде, где набор исходных данных собирается и подается в SWF в виде пакета, который и обрабатывается в рамках соответствующего потока работ [45]. То есть, вычислительный процесс часто планируется на те узлы, где имеется пакет необходимых входных данных [98]. Действия, составляющие SWF в этом случае сильно связаны друг с другом из-за сложных зависимостей между ними. Таким образом, SWF делится на несколько фаз и выполняется последовательно [122]. Например, SWfMS Pegasus может управлять выполнением приложения, выраженного в виде SWF, путем отображения SWF на доступные вычислительные ресурсы и выполнения задач потока работ в порядке их зависимостей [102]. Кроме того, действия SWF могут генерировать большое количество промежуточных данных в течение жизненного цикла SWF [122]. В такой сильносвязанной архитектуре, интенсивная передача данных между действиями SWF может вызвать значительное затруднение в процессе выполнения [72]. Таким образом, для управления и эффективного исполнения

SWF необходимо учитывать особенности организации вычислительных процессов данного типа, включая ограниченное количество доступных ресурсов и возможности планирования с учетом размещения результирующих и промежуточных данных [47].

1.5. Обзор работ по теме диссертации

Задача декомпозиции потоков работ является одной из актуальных задач управления и планирования SWF. Например, в статье [51] предложен подход разделения задачи планирования потоков работ на две подзадачи. Первая – распределение потоков работ по центрам обработки данных, вторая – распределение задач каждого потока работ по вычислительным ресурсам внутри каждого центра обработки данных. Предлагаемое решение обеспечивает улучшение при планировании потока работ в распределенных центрах обработки данных. Однако, при этом сам поток работ остается сильносвязанным, и выполняется в пакетном режиме.

В работе [103] предлагается формальный подход к пошаговой динамической декомпозиции централизованного потока работ на так называемые фрагменты. Он может быть применен непосредственно в процессе выполнения потока работ, обеспечивая возможность миграции фрагментов на независимые вычислительные сервера. Применяя концепцию сетей Петри, фрагмент определяется как часть потока работ, которая состоит из исходного перехода, всех переходов, достижимых из исходного перехода, и всех позиций, связывающих эти переходы. Таким образом, если нам дана сеть Петри $W = (P, T, A)$, которая является WF-net, то фрагментом F называют сеть Петри (P_f, T_f, A_f) такую, что:

1) $T_f \subseteq T; P_f = {}^\bullet T_f \cup T_f{}^\bullet; A_f = A \cap ((P_f \times T_f) \cup (T_f \times P_f))$;
2) F имеет специальный переход t_s такой, что ${}^\bullet({}^\bullet t_s) = \emptyset$;
3) $\forall t \in T_S \; \exists R(t_s, t) = true$.

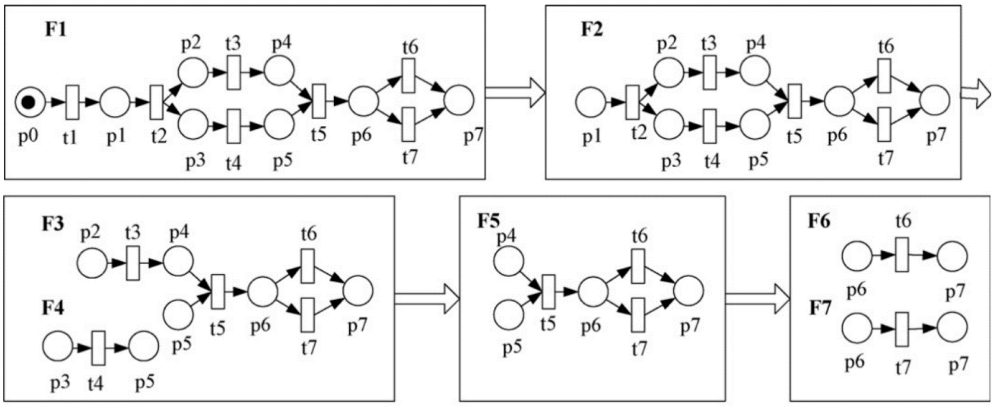


Рис. 11. Процесс динамической фрагментации [103].

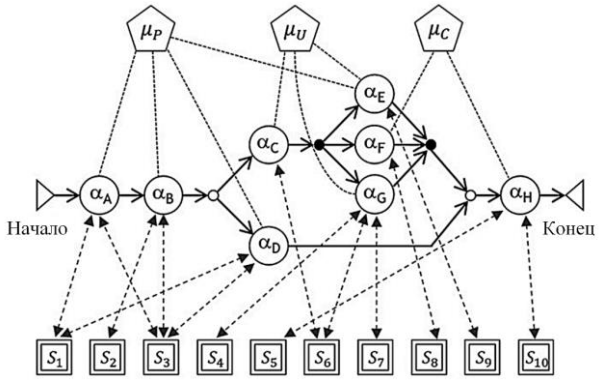


Рис. 12. Пример компонентов модели, предложенной в [4].

Однако особенностью подхода динамической фрагментации является то, что распределение фрагментов на вычислительные узлы может произойти только после успешного выполнения предыдущих фрагментов (см. рис. 11).

Авторами работы [4] предложен формальный подход к решению задачи горизонтальной и вертикальной декомпозиции широкомасштабных потоков работ с учетом возможностей размещения фрагментов потока работ на базе частных, общественных и публичных облачных систем. На рис. 12 показан пример основных компонентов моделей, предложенных. Авторами предлагается следующая модель *потока работ, поддерживающего разбиение (Partial Workflow Model – PWM*):

$$M = (A, S, D, \delta, \lambda, \tau), \quad (7)$$

где:

– A – конечное множество активностей.

– S – конечное множество приложений, обеспечивающих реализацию активностей потоков работ.
– D – конечное множество типов облачного развертывния.
– $\delta = \delta_i \cup \delta_o$, где $\delta_i : A \to \mathcal{P}(A)$ – многозначная функция, отображающая действие $\alpha \in A$ на множество действий, непосредственно предшествующих α, а $\delta_o : A \to \mathcal{P}(A)$ – многозначная функция, отображающая $\alpha \in A$ на множество действий, непосредственно следующих за α.
– $\lambda = \lambda_s \cup \lambda_a$, где $\lambda_s : A \to \mathcal{P}(S)$ – многозначная функция, которая отображает действие $\alpha \in A$ на множество приложений, вызываемых для выполнения α, а $\lambda_a : S \to \mathcal{P}(A)$ – это многозначная функция, которая отображает приложение $s \in S$ на множество действий, которые вызывают s во время их выполнения.
– $\tau = \tau_d \cup \tau_a$, где $\tau_d : A \to \mathcal{P}(D)$ – многозначная функция, которая отображает действие $\alpha \in A$ на набор возможных типов развертывания, а $\tau_a : D \to \mathcal{P}(A)$ – многозначная функция, которая отображает тип развертывания $\mu \in D$ на набор действий, связанных с μ.

Модель фрагмента потока работ определяется как:

$$M_F = (A, S, D, E, \gamma, \chi) \quad (8)$$

где:

– A, S, D – это множества активностей, приложений и типов облачного развертывния соответственно;
– $E \subseteq (D \times D)$ это множество ребер, связывающих типы развертывания, где каждое ребро $(\mu_a, \mu_b) \in E$ отражает

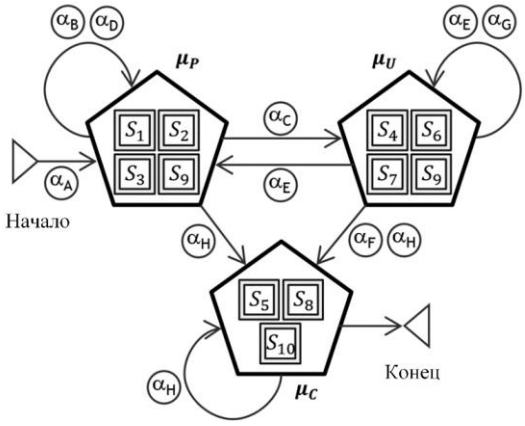


Рис. 13. Пример модели фрагмента потока работ в [4].

существование ребра от активности в развертывании μ_a к активности в развертывании μ_b.

– $\gamma = \gamma_s \cup \gamma_d$, где $\gamma_s : D \to \mathcal{P}(S)$ – многозначная функция, которая отображает тип развертывания $\mu \in D$ на множество приложений, размещенных в облачной среде μ, а $\gamma_d : S \to \mathcal{P}(D)$ – многозначная функция, которая отображает приложениея $s \in S$ на множество типов развертывания $D_s \subseteq D$, в том случае когда s развернуто на множестве соовтетствующих типов развертывания D_s.

– $\chi = \chi_a \cup \chi_e$, где $\chi_a : E \to \mathcal{P}(A)$ – это многозначная функция отображения ребра $(\mu_a, \mu_b) \in E$ на множество активностей, которые связывают μ_a и μ_b, в то время как $\chi_e : A \to \mathcal{P}(E)$ – это многозначная функция отображения от активности $\alpha \in A$ на множество ребер, на которых расположена α.

Авторами предложен алгоритм преобразования PWM в модель фрагмента потока работ (см. алг. 1). На рис. 13 показана модель фрагмента потока работ, созданная в результате применения алгоритма на PWF.

Авторами работы [113] предложено решение, позволяющее интегрировать платформу обработки больших данных Hadoop с SWfMS Kepler для

Алг. 1. Алгоритм преобразования $\boldsymbol{PWM}$ в модель фрагмента потока работ [4]

Вход: $PWM\ \ M = (A, S, D, \delta, \lambda, \tau)$
Выход: Модель фрагмента потока работ, $M_F = (A, S, D, E, \gamma, \chi)$

procedure
  $E \leftarrow \emptyset$;
  for $\forall \alpha \in A$ **do**
    $\gamma_s(\tau_d(\alpha)) \leftarrow \gamma_s(\tau_d(\alpha)) \cup \lambda_s(\alpha)$;
    for $\forall s \in \lambda_s(\alpha)$ **do**
      $\gamma_d(s) \leftarrow \gamma_d(s) \cup \tau_d(\alpha)$;
    end for
    $E \leftarrow E \cup (\tau d(\alpha), \tau d(\delta o(\alpha)))$;
    $\chi_a(\tau_d(\alpha), \tau_d(\delta_o(\alpha))) \leftarrow \chi_a(\tau_d(\alpha), \tau_d(\delta_o(\alpha))) \cup \alpha$;
    $\chi_e(\alpha) \leftarrow \chi_e(\alpha) \cup (\tau_d(\alpha), \tau_d(\delta_o(\alpha)))$;
  end for
end procedure

обеспечения решения задач MapReduce. Показано, что платформа Kepler может упростить управление задачами такого рода за счет графического пользовательского интерфейса, возможностей повторного и совместного использования компонентов. В работе представлен составной актор Kepler MapReduce Actor, который состоит из двух подпотоков: Map и Reduce. Во время инициализации MapReduce Actor, входные данные, хранящиеся в локальной файловой системе, передаются в HDFS, которая затем разбивает и распределяет данные по узлам-рабочим. После этапа ввода данных система исполнения Kepler и подпотоки Map/Reduce распределяются между узлами-рабочими для выполнения задач Map/Reduce на блоках данных, расположенных на этих узлах. После завершения обработки данных на узлах-рабочих Hadoop, MapReduce Actor переносит выходные данные из HDFS в локальную файловую систему, указанный в порту выходного пути (см. рис. 14).

Еще одним примером использования платформы Kepler в интеллектуальном анализе данных могут служить результаты, представленные в статье [58]. Авторы статьи применяют платформу Kepler для решения задачи оптимизации температуры в коммерческой промышленной печи для производства газообразного водорода. Они используют потоки работ Kepler для

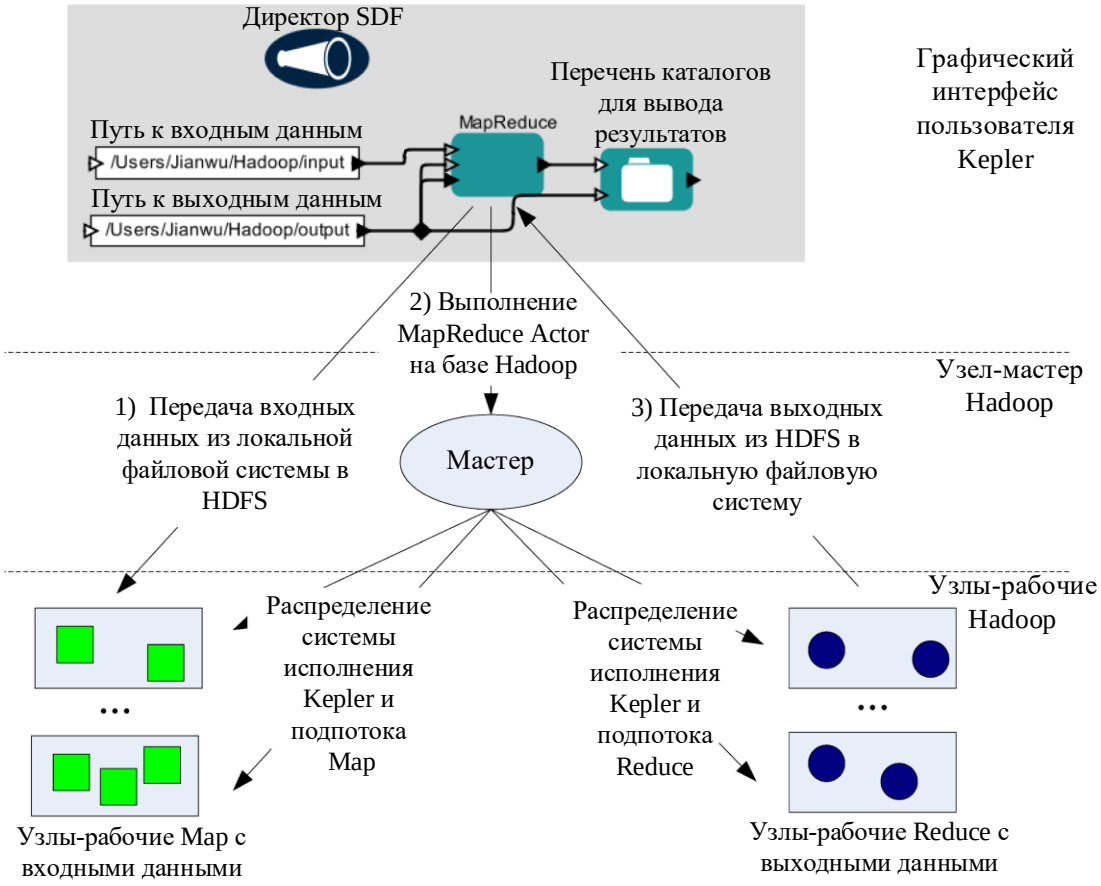


Рис. 14. Архитектура выполнения MapReduce Actor [113].

объединения возможностей пакетов MATLAB и ANSYS для моделирования задач в области вычислительной гидродинамики (*Computational Fluid Dynamics – CFD*).

Однако в работах [4,58,103,113] потоки работ остаются сильно-связанными, в том смысле что отдельные задачи и подпотоки работ жестко связаны между собой. Также, во всех разобранных примерах потоки работ реализуются в формате пакетной обработки данных, что не позволяет применить эти решения в контексте событийно-управляемой архитектуры для поддержки систем IoT в туманных вычислительных средах.

При организации обработки потоков данных IoT в туманных вычислительных системах для таких приложений как DT, необходимо организовать архитектуру системы обработки данных. В дополнение к фундаментальным факторам, таким как объем и тип данных, на сложность организации

управления данными влияют факторы, связанные со свойствами вычислительной среды, поддерживающей DT. Пример решения такой задачи представлен в статье [101] представлен подход к обеспечению краткосрочного прогнозирования нагрузки и обнаружению выбросов в потоках данных, генерируемых интеллектуальными датчиками измерения энергии на основе хранилища Redis [149] и платформы Apache Storm [150]. Для поддержки передачи данных авторы использовали библиотеку обмена сообщений основе кольцевого буфера LMAX.

В статье [21] где предложена «Циклическая архитектура» обработки данных IoT, основанная на комбинации данных от сетей датчиков и распределенных систем обработки событий. Авторами выделяется три слоя в предложенной архитектуре:

– *Слой обмена сообщениями:* для организации ввода/вывода и для организации связи с другими слоями.
– *Слой обработки:* для обработки потока событий.
– *Слой кэширования (энергонезависимый слой):* для кэширования обновленных результатов и связи со внешними системами.

Циклическая архитектура предлагается для обеспечения долгосрочного прогнозирования нагрузки и обнаружения отклонений в высокоскоростном потоке данных от интеллектуальных датчиков измерения энергии.

В статье [41] предложена двухуровневая система обработки потоков данных, в которой центральный уровень данных, основанный на подходе «публикации-подписки» с использованием брокера обмена сообщениями MQTT, обеспечивает связь физического объекта, цифрового двойника и управляющей панели на базе веб-технологий, интегрированной с системой CAD.

Решения, предложенные в работах [21,41,101] не фокусировалась на потребностях туманных вычислений, которые включают необходимость

географического распределения, а также необходимость слабосвязанной архитектуры не только между данными и обработкой, но и в самом слое обработки, где каждый вычислительный объект может быть реализован как независимый сервис. Кроме того, обработка данных в этих статьях реализуется на базе типовых решений обработки потоков данных, таких как Spark, Storm и Kafka без использования систем управления научными потоками работ. Это может привести к сложностям при проектировании и модернизации сложных конфигураций систем обработки данных.

1.6. Выводы по главе 1

Развитие высокопроизводительных вычислительных систем и облачных вычислений обеспечило новые возможности обработки данных, поступающих от интеллектуальных датчиков, и привело к стремительному росту внедрения технологий интернета вещей. Однако на примере таких ключевых приложений IoT как цифровые двойники можно видеть, что облачные вычисления не позволяют обеспечить необходимый уровень обслуживания в части минимизации латентности и обработки данных в непосредственной физической близости от источников данных. Для решения этой проблемы предлагается концепция туманных вычислений, обеспечивающая размещение вычислительных ресурсов между краем сети и централизованным облаком. Существенными особенностями организации таких систем является применение событийно-ориентированной архитектуры и слабосвязанной микросервисной модели организации вычислительных сервисов при обработке потоков данных от систем IoT. Однако системы управления потоками данных обычно не накладывают ограничений на организацию процесса обработки и преобразования, реализуемых компонентами обработки данных. Это может привести к сложностям при проектировании и модернизации сложных конфигураций систем обработки данных. Сегодня, наиболее развитой концепцией для решения подобного класса задач является подход представления

обработки данных в виде так называемых научных потоков работ. Обзор работ, связанных с организацией декомпозиции потоков работ показал, что сегодня отсутствуют решения интеграции событийно-ориентированного слабосвязанного подхода в вычислительные процессы научных потоков работ, в то время как фрагментация потоков работ в пакетном режиме не позволяет обеспечить динамическое масштабирование частей потока работ, а также не позволяет организовать прозрачную интеграцию потоков данных от систем IoT. В связи этим, разработка моделей, методов и алгоритмов обработки потоков данных в туманных вычислительных средах является актуальной задачей.

ГЛАВА 2. МИКРО-ПОТОКИ РАБОТ

В данной главе описывается новая концепция декомпозиции сильносвязанных (раздел 1.2.1) научных потоков работ на наборы слабосвязанных меньших потоков работ, получивших название *«микро-потоки работ»*.

2.1. Концепция микро-потоков работ

Как было показано в обзоре работ по тематике исследования, представленных в разделе 1, несмотря на то что системы управления научными потоками работ предоставляют чрезвычайно удобный механизм для формализации цепочек вычислительных задач, сегодня отсутствуют универсальные решения, которые позволили бы прозрачно интегрировать их в туманные вычислительные системы, поддерживающие обработку потоков данных, поступающих от устройств IoT. Для решения этой задачи предлагается новая концепция, основанная на декомпозиции SWF на наборы слабосвязанных меньших потоков работ, называемых *микро-потоками работ (Micro-Workflow – MWF)*. Данное название образовано из объединения названий двух концепций: микросервисов и потоков работ.

Назовем *монолитным SWF* научный поток работ, спроектированный для обработки данных в пакетном режиме, и состоящий из множества сильно-связанных между собой вычислительных задач.

В отличие от монолитного SWF, каждый MWF представляет собой отдельный вычислительный сервис (микросервис, см. раздел 1.2.2), который может быть разработан, развернут, перенесен на другой вычислительный узел и/или остановлен независимо от других MWF. Связь между MWF обеспечивается посредством событийно-ориентированного подхода, что поддерживает преобразование исполнение SWF из режима пакетной обработки данных в режим потоковой обработки данных (см. рис. 15).

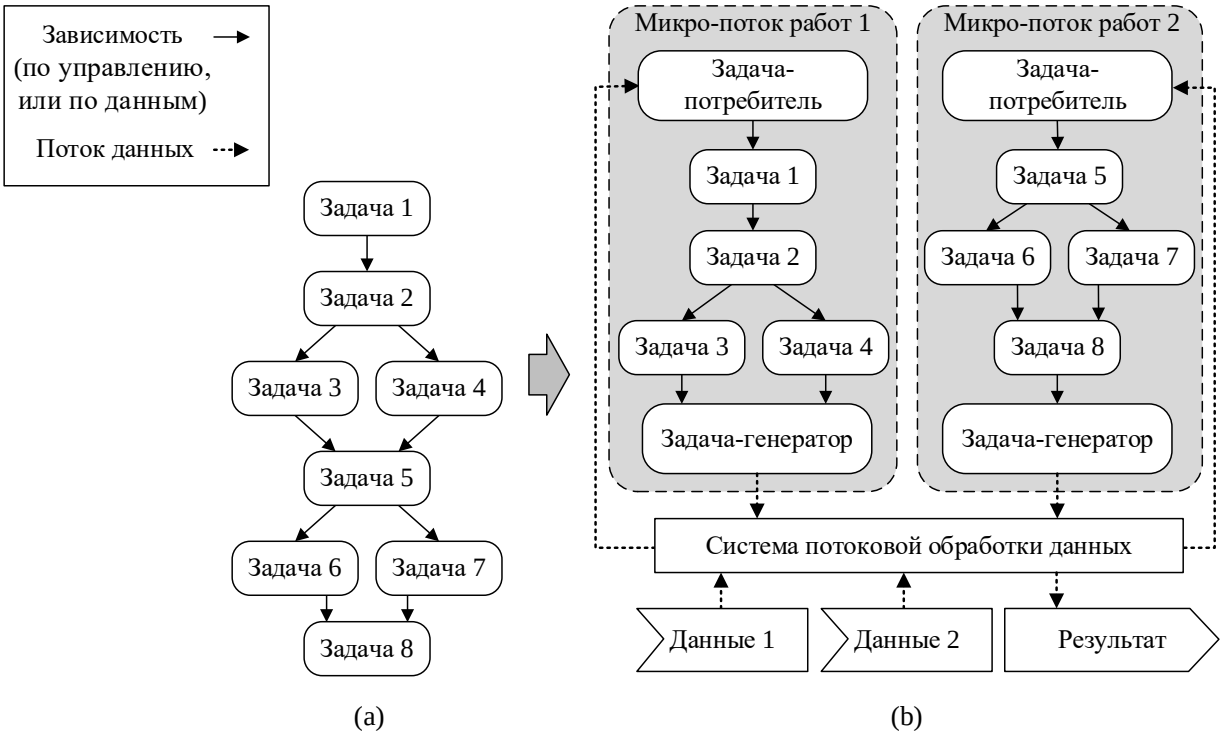


Рис. 15. Концепция микро-потоков работ: (a) монолитный поток работ, (b) микро-потоки работ в результате рефакторинга монолитного потока работ.

Использование этой концепции для разделения монолитного потока работ на набор независимых вычислительных сервисов микро-потоков работ дает следующие преимущества:

- обеспечивается возможность организации работы потока работ в режиме поточной обработки данных;
- развязка сильно связанного вычислительного процесса во времени и пространстве, переход к модели асинхронной коммуникации;
- обеспечивается возможность независимой разработки, обновления, развертывания и запуска MWFs, составляющих общий поток работ, при условии сохранения единого формата обмена сообщениями;
- обеспечивается возможность прозрачной интеграции потоков данных, поступающих от устройств IoT в качестве источников данных для потока работ на любом этапе вычислительного процесса;
- обеспечивается возможность развертывания MWFs общего потока работ на узлах, находящихся на различных уровнях иерархии

туманной вычислительной системы: набор MWFs, которые должны обеспечивать низкую латентность, могут быть развернуты в туманных узлах вблизи устройств IIoT, в то время как другие MWFs, требующие длительной обработки, могут быть перенесены в облако;

– обеспечивается возможность переноса вычислительного процесса MWF с одного вычислительного узла на другой при необходимости перераспределения вычислительной нагрузки без потери промежуточных данных и без необходимости повторной обработки предыдущих данных.

Можно выделить следующие ключевые стадии в процессе рефакторинга монолитного потока работ во множество MWF:

1) разделение монолитного потока работ на под-потоки работ, каждый из которых содержит часть вычислительных задач базового потока работ;
2) формирование специальных вершин потребителей и генераторов сообщений, обеспечивающих связь микро-потоков работ с платформой обработки потоков данных;
3) формирование подразделов (хранилищ) в платформе обработки потоков данных, ответственных за организацию получения и передачи сообщений, генерируемых микро-потоками работ;
4) генерация контейнеров, обеспечивающих инкапсуляцию и возможность независимого развертывания микро-потоков работ в виде микросервисов.

В следующих разделах концепция микро-потоков работ и процесс рефакторинга монолитного потока работ в набор микро-потоков работ рассматриваются более подробно.

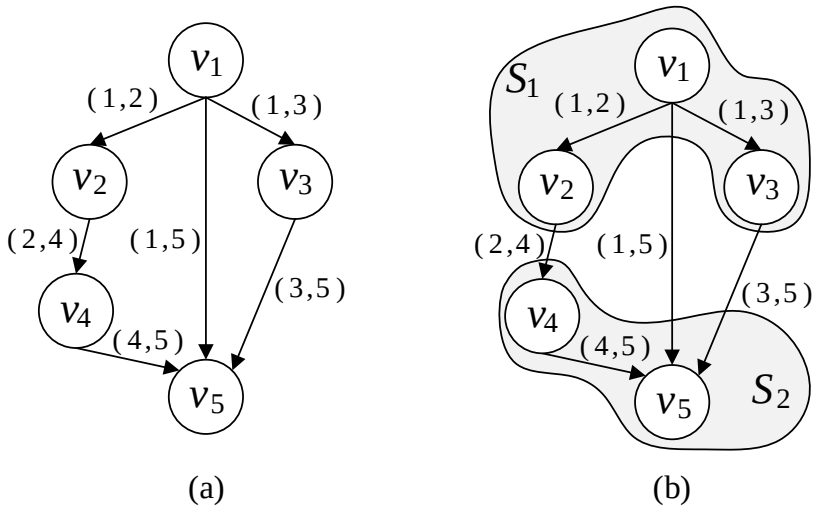


Рис. 16. (a) Пример потока работ W где n принимается равным 5,
(b) пример потока работ W разделенного на два подпотока работ S_1 и S_2.

2.2. Модель микро-потоков работ

2.2.1. Монолитный поток работ

Поток работ может быть представлен в виде DAG (раздел 1.4.2):

$$W = (V, E), \tag{9}$$

где:

– W – монолитный поток работ;

– V – множество вершин, представляющих вычислительные задачи;

– E – множество направленных ребер, соединяющих вершины, представляющих собой зависимости по данным между задачами.

Каждая вершина v_i в V может иметь входные и/или выходные ребра. *Выходная степень* $deg^+(v_i)$ вершины v_i в W определяется как количество ребер, исходящих из v_i и направленных к другим вершинам. *Входная степень* $deg^-(v_i)$ вершины v_i в W определяется как количество ребер, направленных к v_i от других вершин. Ребро $(v_i, v_j) \in E$ представляет собой зависимость по данным от v_i к v_j. Пусть n – это общее количество вершин в V. На рис. 16a приведен пример потока работ W, где n принимается равным 5.

2.2.2. Подпотоки работ

Поток работ W разделяется на множество, состоящее из k *подпотоков работ (Subworkflows)* $S = (S_1, \dots S_k)$, $S_i = (V_i, E_i)$, когда V_i – это множество вершин, входящих в подпоток работ S_i, E_i – это множество ребер между вершинами, входящими в V_i, при этом множество вершин W разделяется подпотоками работ на набор непересекающихся подмножеств:

1) $\forall i \in 1..k: V_i \subset V, E_i \subset E$;
2) $\forall v \in V, \exists i \in 1..k : v \in V_i$;
3) $E_i = \{(v_k, v_l) \in E: v_k, v_l \in V_i\}$;
4) $\forall i, j : i \neq j \implies V_i \cap V_j = \emptyset$.

Рис. 16b показывает пример потока работ W разделенного на два подпотока работ S_1 и S_2. Определим следующие классы ребер и вершин, связанных с подпотоком работ S_i:

1) EI_i: набор входных ребер с начальной вершиной вне подпотока работ S_i и конечной вершиной внутри подпотока работ S_i:

$$EI_i = \{(v_k, v_l) \in E: v_k \notin S_i, v_l \in S_i\}; \quad (10)$$

2) EO_i: набор выходных ребер с начальной вершиной внутри подпотока работ S_i и конечной вершиной вне подпотока работ S_i:

$$EO_i = \{(v_k, v_l) \in E: v_k \in S_i, v_l \notin S_i\}; \quad (11)$$

3) VI_i: набор вершин в S_i, расположенных на головной части ребер EI_i, а также вершин, не имеющих входных ребер:

$$VI_i = \{v_l \in S_i: (v_k, v_l) \in EI_i\} \cup \{v \in S_i: deg^-(v) = 0\}; \quad (12)$$

4) VO_i: набор вершин в S_i, расположенных на концах ребер EO_i, а также вершин, не имеющих выходных ребер:

$$VO_i = \{v_k \in S_i: (v_k, v_l) \in EO_i\} \cup \{v \in S_i: deg^+(v) = 0\}. \quad (13)$$

2.2.3. Определение микро-потока работ

Чтобы преобразовать подпоток работ S_i в микро-поток работ MWF_i, необходимо извлечь все вершины S_i из потока работ W и обеспечить коммуникационные механизмы, связывающие MWF_i с платформой потоковой передачи событий через *вершину-потребитель* (*consumer vertex* – cv_i) и *вершину-генератор* (*producer vertex* – pv_i).

Вершина cv_i в MWF_i обеспечивает потребление потока входных данных от платформы потоковой передачи данных (раздел 1.3.1) и распределяет его между вершинами VI_i. Вершина pv_i в MWF_i действует как сток, который собирает выходные данные из вершин, составляющих множество VO_i и передает их в виде сообщений на платформу потоковой передачи данных. Определим соответствующие множества ребер следующим образом:

– $ECV_i = \{(cv_i, v): v \in VI_i\}$, набор ребер, идущих от cv_i к вершинам в VI_i;
– $EPV_i = \{(v, cp_i) : v \in VO_i\}$, набор ребер, идущих от вершин в VO_i к cp_i.

В этом случае микро-поток работ MWF_i из подпотока работ S_i определяется как:

$$MWF_i = (MV_i, ME_i), \tag{14}$$

где:

– $MV_i = V_i \cup \{cv_i, pv_i\}$ – множество всех вершин, находящихся внутри S_i, включая cv_i и pv_i;
– $ME_i = E_i \cup ECV_i \cup EPV_i$ – множество всех ребер, расположенных внутри S_i, включая все ребра, которые идут из cv_i к вершинам в VI_i, а также все ребра, идущие от вершин в VO_i к cp_i.

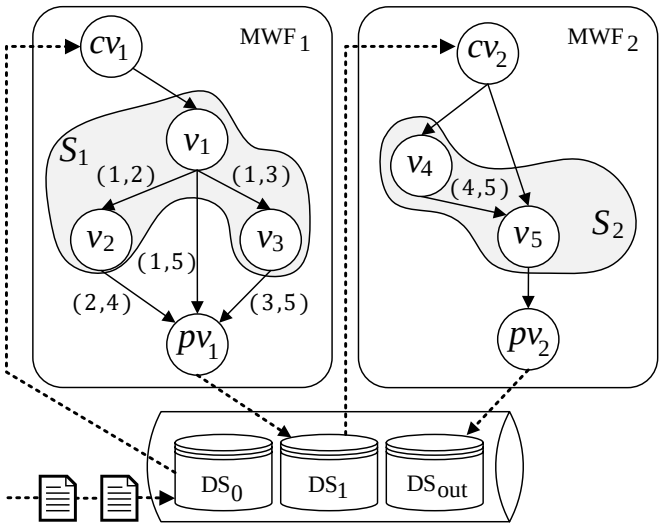


Рис. 17. Результат применения модели микро потока работ.

Рис. 17 показывает набор микро-потоков работ, полученных в результате извлечения подпотоков работ из монолитного потока работ W, показанного на рис. 16а.

2.2.4. Обработка потоков данных в модели микро-потоков работ

Модель микро-потоков работ объединяет модели потоков работ и потоковой обработки данных. В структуре платформы потоковой обработки данных формируется набор выделенных каналов (хранилищ потоков данных) для организации взаимодействия микро-потоков работ посредством обмена сообщениями. Каждое сообщение является набором данных, который включает в себя:

– метку времени создания сообщения,
– информацию об источнике данных,
– структурированную коллекцию передаваемых данных.

Необходимы следующие хранилища потоков данных (рис. 17):

– DS_0 отвечает за сбор, хранение и предоставление сообщений, содержащих наборы данных, необходимых для инициализации вычислительного процесса в потоке работ;

– DS_i отвечает за получение сообщений, содержащих промежуточные данные, от MWF_i, и передачу их зависимым микро-потокам работ;
– DS_{out} отвечает за сбор, хранение и предоставление сообщений, содержащих результирующие данные.

Вместо начального узла потока работ, наборы данных, необходимые для запуска вычислительного процесса, поступают в хранилище потоков данных DS_0 в виде сообщений. Обработка сообщений из хранилища потока данных организована следующим образом:

1) cv_i соответствующего MWF_i извлекает последующее сообщение из DS_{i-1}.
2) На основе анализа полученного сообщения cv_i инициирует передачу данных по ребрам ECV_i к вершинам, ответственным за выполнение непосредственного вычислительного процесса.
3) После завершения задач обработки данных и отправки данных по ребрам EPV_i, pv_i генерирует исходящее сообщение в DS_i или DS_{out}, если это окончательный результат.

2.3. Алгоритм рефакторинга микро-потоков работ

В данном разделе представлен алгоритм, обеспечивающий рефакторинг монолитного потока работ в набор микро-потоков работ. В алгоритме рефакторинга могут быть выделены следующие ключевые шаги:

1. Инициализация матрицы Z для представления монолитного потока работ W и множества подпотоков S, на которые разбивается поток работ W.
2. Отображение в матрице Z внутренних и внешних ребер для подпотоков работ $S_x \in S$.
3. Создание матрицы M_x для представления каждого микро-потока работ MWF_x. Для каждого подпотока работ $S_x \in S$:
   3.1.Инициализация матрицы M_x.

3.2. Отображение множества внутренних ребер подпотока E_x в соответствующую матрицу M_x.

3.3. Формирование множества ребер, исходящих из вершины-потребителя cv_x и их отображение в M_x.

3.4. Формирование множества ребер, входящих в вершину-генератор pv_x и их отображение в M_x.

Для описания монолитного потока работ $W = (V, E)$, разделенного на набор подпотоков работ $S = (S_1, \dots S_k)$, определим матрицу $Z_{n\times n}$, где n – это общее количество вершин в потоке работ:

$$Z = \begin{bmatrix} z_{1,1} & \cdots & z_{1,n} \\ \vdots & \ddots & \vdots \\ z_{n,1} & \cdots & z_{n,n} \end{bmatrix}. \quad (15)$$

В основе предложенного подхода лежит модификация механизма представления графов в виде матрицы смежности (раздел 1.4.2). Матрица Z будет представлять информацию о ребрах между вершинами и о том, к какому подпотоку принадлежит каждая вершина. Инициализация этой матрицы реализуется следующим образом:

$$z_{i,j} = \begin{cases} 1, & i \neq j \ \wedge \ (v_i, v_j) \in E \\ 0, & i \neq j \ \wedge \ (v_i, v_j) \notin E, \\ x, & i = j \ \wedge \ v_i \in V_x \end{cases} \quad (16)$$

где

– $i, j \in 1..n$ – индексы элементов матрицы Z;

– $x \in 1..k$ – это индекс подпотока $S_x = (V_x, E_x)$.

Значение $z_{i,i}$ на диагонали матрицы в этом случае соответствует индексу x подпотока работ S_x, которому принадлежит вершина с индексом i, в то время как остальные значения элементов матрицы $z_{i,j}$ обозначают собой наличие ребра между вершинами v_i и v_j , где 0 означает отсутствие ребра между v_i и v_j, а «1» означает наличие соответствующего ребра.

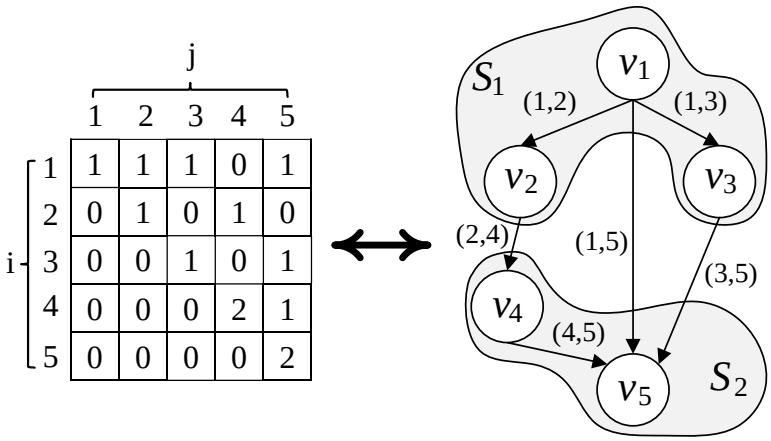


Рис. 18. Пример инициализации матрицы Z полученный по формуле (16) для примера потока работ W из двух подпотоков.

На рис. 18 показан пример инициализации матрицы Z для примера потока работ W из двух подпотоков.

На следующем шаге алгоритма происходит определение внутренних и внешних ребер для каждого подпотока в Z:

$$z_{i,j} = \begin{cases} z_{i,i}, & i \neq j \ \wedge \ z_{i,j} = 1 \ \wedge \ z_{i,i} = z_{j,j} \\ -1, & i \neq j \ \wedge \ z_{i,j} = 1 \ \wedge \ z_{i,i} \neq z_{j,j}. \\ z_{i,j}, & \text{иначе} \end{cases} \quad (17)$$

Так как на шаге инициализации (16) в значении элементов матрицы Z, расположенных на диагонали $(z_{i,i})$ был сохранен индекс подпотока, в которой находится вершина v_i, эти значения будут использованы в формуле (17) для распознавания внутреннего и внешнего ребер для каждого подпотока в Z. В первом случае определяется множество внутренних ребер, которые связывали две вершины v_i и v_j находящиеся в одном и том же подпотоке. Это реализуется путем сопоставления соответствующих диагональных значений $z_{i,i}$ и $z_{j,j}$ для ребра (v_i, v_j), представленного в $z_{i,j} = 1$.

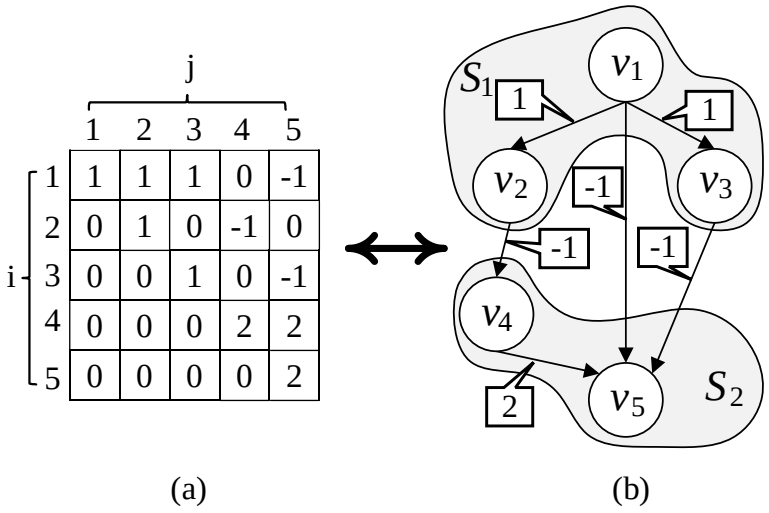


Рис. 19. Внутренние и внешние ребра подпотоков: a) матрица Z после выполнения шага (17), b) визуализация W после выполнения шага (17).

Если значения $z_{i,i}$ и $z_{j,j}$ для ребра, представленного $z_{i,j}$ одинаковы, это означает, что вершины v_i и v_j находятся в одном и том же подпотоке, и значение $z_{i,j}$ устанавливается равным значению индекса соответствующего подпотока, сохраненного в соответствующих диагональных элементах матрицы Z.

Во втором случае, если значение соответствующих диагональных элементов $z_{i,i}$ и $z_{j,j}$, характеризующих вершины v_i и v_j, различно, то вершины находятся в разных подпотоках, и (v_i, v_j) – это внешнее ребро. В этом случае значение ребра $z_{i,j}$ устанавливается равным «-1». На рисунке 19a показан пример применения шага, описанного в формуле (17). Результатом этого шага является сформированная матрица Z.

На следующем шаге алгоритма происходит инициализация матриц M_x представления микро-потоков работ MWF_x на основе подпотоков S_x. Определим M_x как матрицу, используемую для определения микро-потока работ MWF_x на основе подпотока S_x и вершин cv_x и pv_x. Во избежание путаницы, для индексации строк и столбцов элементов в матрицах представления микро-потоков M_x будем использовать соответственно переменные a и b (см. рис. 20).

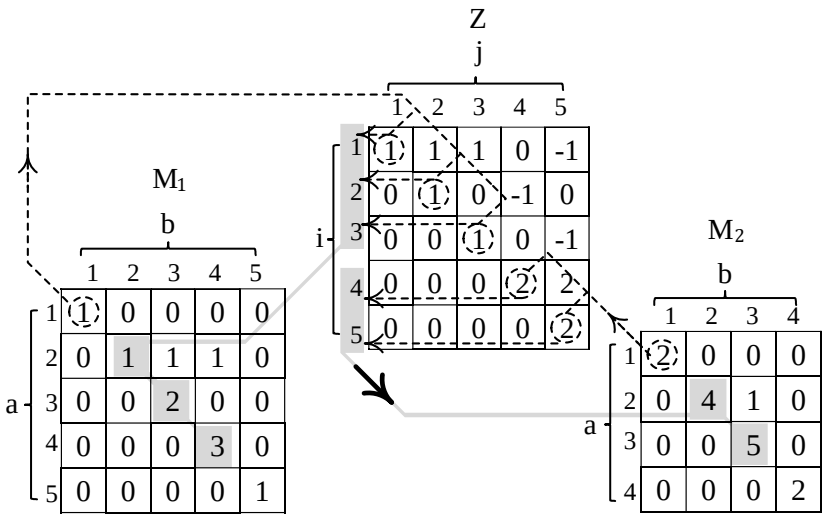


Рис. 20. Матрица Z, и инициализация матриц M_1 и M_2.

Размерность r_x матрицы M_x определяется количеством строк в матрице Z, у которых значения на диагонали равны x:

$$r_x = \left|\{a \in 1..n: z_{a,a} = x\}\right|.$$

Для представления вершины-потребителя (cv_x) и вершины-генератора (pv_x) необходимо в матрицу M_x добавить еще 2 строки и 2 столбца:

$$M_x = \begin{bmatrix} m_{x_{1,1}} & \cdots & m_{x_{1,r_x+2}} \\ \vdots & \ddots & \vdots \\ m_{x_{r_x+2,1}} & \cdots & m_{x_{r_x+2,r_x+2}} \end{bmatrix}. \quad (18)$$

Представление узла cv_x будет размещено на позиции $m_{x_{1,1}}$, а pv_x – на позиции $m_{x_{r_x+2,r_x+2}}$. Диагональные значения в матрице M_x обоих этих узлов установим равными x. В остальных диагональных значениях сохраним индексы i вершин потока работ W, которые входят в подпоток S_x, т.е. имеют значение, равное x. Сохраним значения индексов i всех вершин из S_x в D_x где x – это индекс подпотока S_x:

$$D_x = \{\forall i \in 1..n: z_{i,i} = x\}. \quad (19)$$

Значения $d \in D_x$ будут использованы в (20) для заполнения диагональных значений для матрицы M_x:

$$m_{x_{a,a}} = \begin{cases} d_{a-1}, & 1 < a < r_x + 2 \\ x, & a = 1 \vee a = r_x + 2 \end{cases}, \quad (20)$$

где:

– $x \in 1..k$ – это индекс подпотока S_x;

– $a \in 1..r_x + 2$ – это индекс строки в матрице M_x.

На рис. 20 показан пример применения (18), (19) и (20) для инициализации матриц микро-потоков M_1 и M_2 из Z.

После инициализации матриц M_x для каждого подпотока S_x, реализуется размещение в них представления всех внутренних ребер подпотоков E_x. Для того, чтобы перенести внутренние ребра E_x из Z в M_x применяется формула (21):

$$m_{x_{a,b}} = \begin{cases} 1, & z_{m_{x_{a,a}}, m_{x_{b,b}}} = m_{x_{1,1}} \\ 0, & \text{иначе} \end{cases}, \quad (21)$$
$$\forall m_{x_{a,b}} \in M_x : a, b \in 2..r_x + 1 \wedge a \neq b.$$

В этом случае применяется информация о смежных вершинах, сохраненная в диагональных позициях M_x в качестве ссылок, где a – индекс строки в M_x, а b – индекс столбца в M_x . На рис. 21а показан пример применения (21) для переноса внутреннего ребра (v_1, v_2) из E_1 в M_1, а также результаты представления внутренних ребер подпотоков работ из множеств E_1 и E_2 в M_1 и M_2 соответственно.

После представления всех внутренних ребер из E_x в M_x происходит определение выходных ребер для вершины-потребителя cv_x в матрицах описания подпотоков M_x. Существует 2 типа вершин в M_x, которые должны получать входные ребра от cv_x:

– начальная вершина W, не имеющая входных ребер;

– вершины, которые получают входные ребра от вершин, находящихся в других подпотоках.

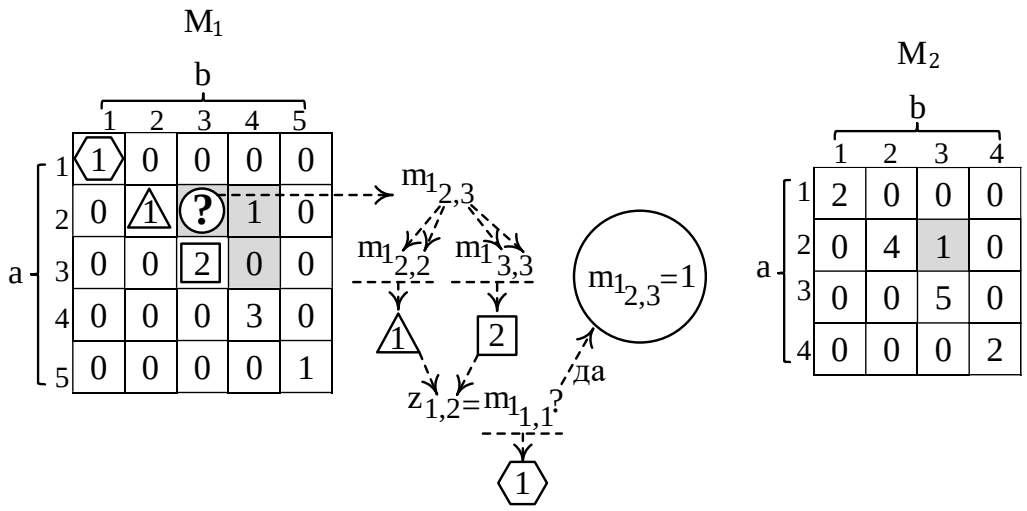


Рис. 21. Результаты применения формулы (21) для представления внутренних ребер из множеств E_1 и E_2 в M_1 и M_2 соответственно.

Формула (22) определяет значение позиций в первой строке матриц M_x, определяющих выходные ребра из вершины cv_x:

$$m_{x_{1,b}} = \begin{cases} 1, \begin{pmatrix} (\forall i \in 1..n \wedge i \neq m_{x_{b,b}} : z_{i,m_{x_{b,b}}} = 0) \\ \vee\ (\exists i \in 1..n : z_{i,m_{x_{b,b}}} = -1) \end{pmatrix}, \\ 0, \text{иначе} \end{cases} \tag{22}$$

$$\forall m_{x_{1,b}} \in M_x : b \in 2..r_x + 1.$$

Первый случай для значения «1» в (22) определяет ребра из cv_x в те вершины, которые изначально не имеют входных ребер в W. Для получения индекса столбца, по которому производится поиск в матрице Z, используется значение $m_{x_{b,b}}$ где $b \in 2 \ldots r_x + 1$. Если значения всех элементов матрицы (кроме диагонального) Z в столбце $m_{x_{b,b}}$ равны 0, это значит, что изначально у вершины с индексом $m_{x_{b,b}}$ не было входных ребер в Z, таким образом она должна быть соединена с cv_x в M_x. На рис. 22 показан пример применения этого случая формулы (22) для определения наличия ребра, которое должно быть представлено в позиции $m_{1_{1,2}}$ в M_1.

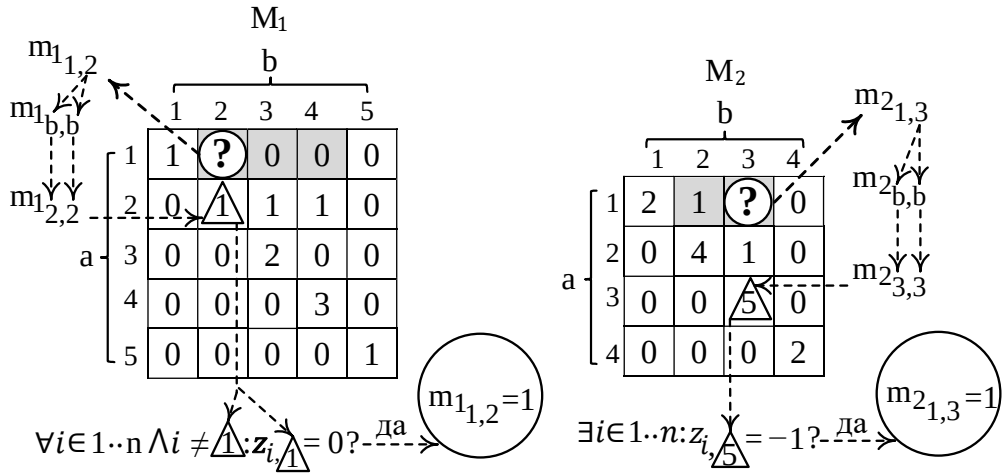


Рис. 22. Применение формулы (22) к M_1 и M_2 для соединения cv.

Второй случай для значения «1» в формуле (22) определяет ребра из cv_x в те вершины, которые имеют входные ребра из других подпотоков. Аналогичным образом, значение $m_{x_{b,b}}$ используется для получения индекса столбца, по которому производится поиск в матрице Z.

Наличие хотя бы одного элемента со значением «-1» в соответствующем столбце означает, что данная вершина получает данные из внешнего подпотока. Данная вершина должна получить входное ребро из cv_x в M_x. На рис. 22 показан пример применения этого случая формулы (22) для определения наличия ребра, которое должно быть представлено в позиции $m_{2_{1,3}}$ в M_2.

Финальным шагом алгоритма рефакторинга является определение ребер, которые должны быть завершаться в вершине-генераторе pv_x для каждой матрицы подпотока M_x. Есть 2 типа вершин в S_x, которые должны быть связаны ребрами с pv_x:

– вершина, не имеющая выходных ребер в E_x;
– вершины, выходные ребра из которых идут к вершинам, находящимся в других подпотоках.

Формула (23) определяет значение позиций в последнем столбце $(r_x + 2)$ матриц M_x, определяющих входные ребра в вершину pv_x:

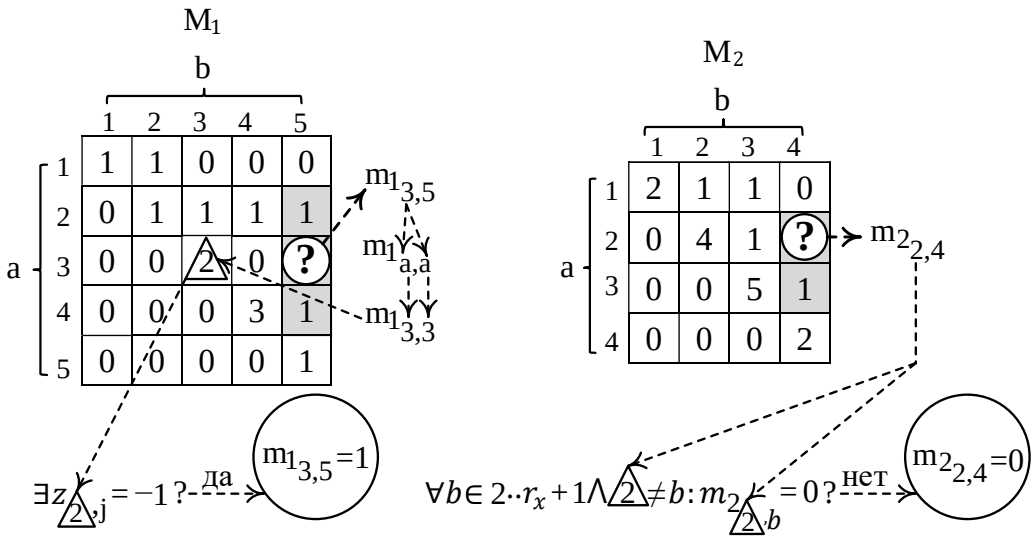


Рис. 23. Применение формулы (23) к M_1 и M_2 для соединения pv.

$$m_{x_{a,r_x+2}} = \begin{cases} 1, \begin{pmatrix} (\forall b \in 2..r_x + 1 \wedge a \neq b: m_{x_{a,b}} = 0) \\ \vee\ (\exists j \in 1..n: z_{m_{x_{a,a}},j} = -1) \end{pmatrix}, \\ 0, \text{иначе} \end{cases} \tag{23}$$

$$\forall m_{x_{a,r_x+2}} \in M_x: a \in 2..r_x + 1.$$

Первый случай для значения «1» в формуле (23), определяет те вершины, которые изначально не имеют выходных ребер. Такие вершины соединяются ребрами с pv_x. На рис. 23 показан пример применения этого случая формулы (23) для определения наличия ребра, которое должно быть представлено в позиции $m_{2_{2,4}}$ в M_2.

Второй случай для значения «1» в формуле (23), определяет ребра в pv_x из тех вершин, которые изначально имели выходные ребра в вершины, содержащиеся в других подпотоках. Значение $m_{x_{a,a}}$ используется для получения индекса строки, по которой производится поиск элементов в матрице Z. Наличие хотя бы одного элемента со значением «-1» в строке с индексом $m_{x_{a,a}}$ в матрице Z означает, что данная вершина передает данные во внешний подпоток. В этом случае, данная вершина должна быть соединена выходным ребром с pv_x в M_x. На рис. 23 показан пример применения этого

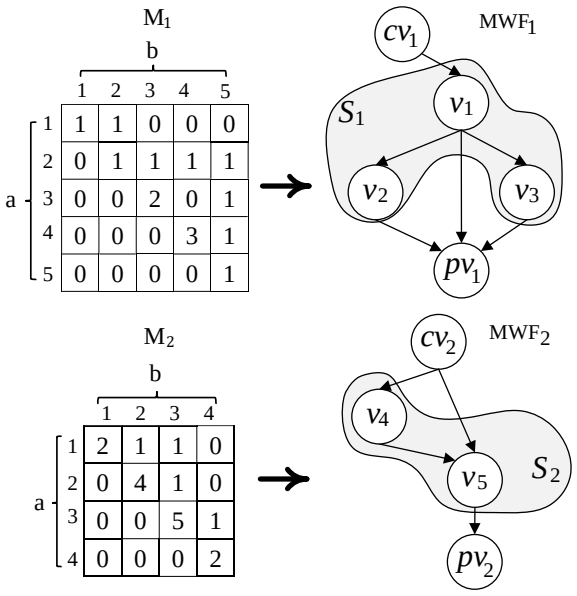


Рис. 24. Финальное состояние матриц M_1 и M_2 представляющих MWF_1 и MWF_2.

случая формулы (23) для определения наличия ребра, которое должно быть представлено в позиции $m_{1_{3,5}}$в M_1.

На рис. 24 показаны финальные состояния матриц M_1 и M_2, представляющих, микро-потоки работ MWF_1 и MWF_2, сформированные на основе разбиения монолитного потока работ W, приведенного в качестве примера на рис. 18.

2.4. Вывод по главе 2

Во второй главе представлена математическая модель микро-потоков работ для обработки потоков данных в распределенных вычислительных средах, таких как туманные вычисления. Модель включает в себя алгоритм рефакторинга зависимостей в монолитном потоке в набор автономных микро-потоков работ. Такое разделение поддерживает независимость реализации, исполнения, разработки, сопровождения и кроссплатформенного развертывания микро-потоков работ на независимых вычислительных узлах. Предлагаемый подход, модель, и алгоритм рефакторинга монолитного потока работ, представленные во второй главе, опубликованы в работах [6,8,10,85,124].

ГЛАВА 3. ПРОГРАММНАЯ ПОДДЕРЖКА МОДЕЛИ МИКРО-ПОТОКОВ РАБОТ

Для поддержки реализации и тестирования модели микро-потоков работ и алгоритмов, представленных в главе 2, на языке Java был разработан комплекс вычислительных акторов для платформы управления потоками работ Kepler (раздел 1.4.3). Разработанные акторы обеспечивают поддержку функционирования микро-потоков работ путем реализации вершины-потребления, вершины-генератора, а также ряда типовых операций обработки потоков данных. С использованием данных акторов, на базе платформы Kepler был разработан набор потоков работ, решающих типовую задачу обработки потоков данных интернета вещей, как в формате монолитного потока работ, так и в формате отдельных микро-потоков работ.

Для обеспечения поддержки потоковой обработки данных, а также в качестве среды для поддержки событийно-ориентированной архитектуры использовалась платформа обработки потоков данных Apache Kafka (раздел 1.3.7), интеграция с которой реализуется посредством набора Kafka API. Контейнеризация и управление развертыванием микро-потоков работ на узлах распределенной вычислительной системы реализуется посредством платформы Docker (раздел 1.3.5). Для проведения вычислительных экспериментов, также был разработан ряд программных утилит, поддерживающих симуляцию IoT потоков работ, репликацию данных между географически-распределенными локациями серверов Apache Kafka, а также рефакторинг монолитных потоков работ. В этой главе представлены ключевые аспекты реализации и функционирования разработанных программных компонентов.

3.1. Акторы Kepler для организации потоковой обработки данных

Для оценки нашего подхода к организации вычислительного процесса в виде микро-потоков работ был разработан набор акторов, расширяющих

возможности системы Kepler для обеспечения потоковой обработки данных с использованием платформы Apache Kafka. В процессе запуска и исполнения, реализованные акторы могут считывать информацию как из переменных среды, значения которых установлены на узле, на котором развернут поток робот, так и из своего графического интерфейса, в случае если переменные среды не установлены. Таким образом, обеспечивается возможность исполнения данных акторов как в интерактивном режиме с поддержкой пользовательского интерфейса, так и в автономном режиме.

3.1.1. Актор KafkaConsumer

Актор ***KafkaConsumer*** представляет собой реализацию вершины-потребителя микро-потока работ (раздел 2.2.4). Актор KafkaConsumer подписывается на тему Kafka, подключаясь к серверу Apache Kafka по протоколу TCP (раздел 1.3.7). После подключения, данный актор получает поток данных (в виде последовательности сообщений) из этой темы. Каждое сообщение содержит набор полей данных, которые стерилизуются с помощью системы сериализации данных Apache Avro (раздел 1.3.1). Актор обеспечивает десериализацию каждого полученного сообщения в набор токенов (тип данных в системе Kepler, см. раздел 1.4.3), которые затем отправляются на выходной порт этого актора. Для поддержки десериализации данных, актору необходима схема Apache Avro Schema для входящих сообщений. Актор получает данную схему с сервера реестра схем. Листинг 1 представляет собой псевдокод процесса работы актора KafkaConsumer. Структура параметров PARAMS актора KafkaConsumer и их описание приведены ниже:

– SOURCE_KAFKA_SERVER («Kafka Address and Port» в Kepler GUI): строка, содержащая адрес и порт сервера Apache Kafka, который является источником данных, например «192.168.1.3:9092».
– SOURCE_SCHEMA_REG_SERVER («Schema Registry Address and Port» в Kepler GUI): строка, содержащая адрес и порт сервера

Листинг 1. Псевдокод актора KafkaConsumer

```
 1: procedure KafkaConsumer

                                                  ▷ Стадия инициализации
 2: if environmentVariables = null
 3:     PARAMS ← GUI
 4: else
 5:     PARAMS ← environmentVariables
 6: con ←ConnectToKafkaConsume(PARAMS.SOURCE_KAFKA_SERVER,
    PARAMS.SOURCE_GROUP_ID, PARAMS.SOURCE_TOPIC,
    PARAMS.SOURCE_CONSUME_TIMEOUT)
 7: schema ←ReadSchema(PARAMS.SOURCE_SCHEMA_REG_SERVER)

                                          ▷ Цикл обработки сообщений
 8: message ←ConsumeMessage(con,Schema)
 9: rcvMsg ←GetRecieveTimestamp(Message)
10: record ←InitializeRecord()
11: if PARAMS.SOURCE_PARMS_TO_READ <> null
12:     for each field in message:
13:         if field.name ∈ PARAMS.SOURCE_PARMS_TO_READ:
14:             record[PARAMS.SOURCE_TOPIC].Insert(field)
15: else
16:     for each field in message:
17:         record[PARAMS.SOURCE_TOPIC].Insert(field)
18: record.Insert(rcvMsg)
19: SendRecordToOutputPort(record)
20: goto 8
21: end procedure
```

регистрации схем данных Apache Avro, который является источником схем сообщений для темы исходных данных Kafka, например «192.168.1.3:8081».

– SOURCE_GROUP_ID («Group Id To Subscribe» в Kepler GUI): строка, содержащся идентификатор для группы потребителей (раздел 1.3.7).
– SOURCE_PARMS_TO_READ («Parameters To Read» в Kepler GUI): используется в качестве опции для чтения определенных полей из потребляемого сообщения. Входящие сообщения потока данных могут содержать большое количество полей данных, информация из которых не требуется для дальней обработки. В этом параметре указываются идентификаторы полей (разделенные пробелами)

которые должны быть десериализованы и отправлены в виде токенов в выходной порт, например «index timestamp».

– SOURCE_TOPIC («Topic To Subscribe» в Kepler GUI): строка, содержащая идентификатор темы источника данных в Apache Kafka из которой будет производиться чтение потока данных.
– SOURCE_CONSUME_TIMEOUT («Timeout» в Kepler GUI): время, в миллисекундах, которое может быть затраченно на ожидание операции сбора данных, если данные в источнике недоступны. Если значение параметра установлено равным 0, то любые записи, доступные на текущий момент времени возвращается немедленно, иначе возвращается *null*.
– firingCountLimit: служебное поле системы Kepler, отвечающее позволяющее ограничить максимальное количество запусков данного актора. Если в поле указано позитивное целое число, то актор будет вызван не более указанного количества раз. Установка данного поля в значение NONE позволяет данному актору исполняться неограниченное число раз. Этот параметр наследуется от стандартного шаблона актора в среде Kepler.

3.1.2. Актор KafkaProducer

Актор ***KafkaProducer*** представляет собой реализацию вершины-генератора микро-потока работ (раздел 2.2.4). Данный актор получает токены Kepler в качестве входных данных на входной порт, сериализует каждую полученную запись токенов в виде сообщения в формате Apache Avro, а затем отправляет это сообщение соответствующей теме Apache Kafka на сервере Kafka по протоколу TCP. Для поддержки сериализации данных, актору необходима схема Apache Avro Schema для исходящих сообщений. Актор получает данную

Листинг 2. Псевдокод актора KafkaProducer

```
 1: procedure KafkaProducer

                                                  ▷ Стадия инициализации
 2: if environmentVariables = null
 3:       PARAMS ← GUI
 4: else
 5:      PARAMS ← environmentVariables
 6: con ← ConnectToKafkaProduce(PARAMS.OUT_KAFKA_SERVER,
    PARAMS.OUT_TOPIC)
 7: schema ← PARAMS.OUT_AVRO_SCHEMA
 8: SendSchema(PARAMS.OUT_SCHEMA_REG_SERVER , schema)

                              ▷ Цикл обработки входящих записей токенов
 9: record ←RecieveRecordFromInputPort()
10: message ←InitializeMessage(schema)
11: if PARAMS.OUT_PARMS_TO_READ <> null
12:       for each field in record:
13:            if field.name ∈ PARAMS.OUT_PARMS_TO_READ:
14:                 message.Insert(field)
15: else
16:       for each field in record:
17:            message.Insert(field)
18: ProduceMessage(con, message)
19: goto 9
20: end procedure
```

схему с сервера реестра схем. Листинг 2 представляет собой псевдокод работы актора KafkaProducer. Структура параметров PARAMS актора и их описание приведено ниже:

– OUT_KAFKA_SERVER («Kafka Address and Port» в Kepler GUI): строка, содержащая адрес и порт сервера Apache Kafka, который является приемником данных, например «192.168.1.3:9092».

– OUT_SCHEMA_REG_SERVER («Schema Registry Address and Port» в Kepler GUI): адрес и порт сервера регистрации схем данных Apache Avro, который является источником схем сообщений для темы целевых данных Kafka , например «192.168.1.3:8083».

– OUT_TOPIC («Topic To Subscribe» в Kepler GUI): идентификатор темы потребителя данных в Apache Kafka в которую будет производиться запись потока данных.
– OUT_AVRO_SCHEMA («Schema» в Kepler GUI): идентификатор схемы, используемой для сериализации выходных данных.
– OUT_PARMS_TO_READ («Parameters To Read» в Kepler GUI): используется в качестве опции для чтения определенных полей из записей токенов, поступающих на входной порт. Входящие записи могут содержать большое количество полей данных, информация из которых не требуется для дальней обработки. В этом параметре указываются идентификаторы полей (разделенные пробелами) которые должны быть сериализованы и отправлены в сообщения Apache Avro на сервер Kafka.
– firingCountLimit: см. описание аналогичного поля актора KafkaConsuper в разделе 3.1.1.

3.1.3. Актор DetectStateChange

Актор ***DetectStateChange*** разработан для постоянного мониторинга изменений в сериях данных, получаемых от отдельного источника данных или от набора источников данных. Рассмотрим список параметров данного актора:

– inTopic («Topic» в Kepler GUI): строка, содержащая перечень идентификаторов тем Kafka, в рамках которых производится отслеживание состояния.
– inPP («Input Sensors Field» в Kepler GUI): строка, содержащая перечень идентификаторов сенсоров во входящем сообщении, которые необходимо отслеживать на предмет изменения состояния.
– inTS («Input Timestamp Field» в Kepler GUI): строка, содержащая идентификатор поля временной метки для соответствующего датчика во входящем сообщении. Значение по умолчанию: «ts».

– inIndex («Msg Index» в Kepler GUI): строка, содержащая идентификатор поля индекса во входящем сообщении. Значение по умолчанию: «index».
– inMsgRcvTime («Msg rcv time» в Kepler GUI): строка, содержащая идентификатор поля, содержащего время получения входящего сообщения. Значение по умолчанию: «rcvTime».
– outEdges («Output edges» в Kepler GUI): имя поля, в котором выдается результат, указывающий на обнаружение изменение состояния во входных записях данных.
– outTS («Output timestamps» в Kepler GUI): имя поля в производимых сообщениях, которое будет содержать временную метку состояния изменения (outEdges).

В рамках данного актора отдельный источник данных обозначается как «тема» (topic). В каждую единицу времени актор получает на вход запись токенов. Каждая полученная запись токенов $Record_{in}$ может быть представлена следующим образом:

$$Record_{in} = \left(topic_{1_{in}}, .., topic_{n_{in}}\right). \quad (24)$$

Идентификатор каждой темы $topic_{i_{in}} \in Record_{in}$ представляет собой идентификатор одного из источников серии данных. Если пользователю требуется обнаружить изменение состояния в определенном наборе источников, он может указать набор интересующих его идентификаторов источников в параметре inTopic, указав их идентификаторы через пробел (например, как «topic1 topic2 topic3»).

Каждая тема $topic_{i_{in}}$ представляет собой структуру данных, состоящую из следующих элементов:

$$topic_{i_{in}} = (index, ts, [p_1 .. p_m], rcvTime) \quad (25)$$

где $index$ — индекс исходного сообщения, полученного от источника, считывается из поля с идентификатором, указанном в значении параметра

inIndex; ts – метка времени считывания данных, считывается из поля с идентификатором указанным в значения параметра inTS; $[p_1..p_m]$ — набор значений, где значение каждого p_i представляет данные, сгенерированные сенсором $p_i \in topic_{i_{in}}$ во время ts; $rcvTime$ – метка времени получения сообщения в системе обработки потоков работ, считывается из поля с идентификатором, указанным в значении параметра inMsgRcvTime.

Если пользователь хочет обнаружить изменение состояния в определенном подмножестве сенсоров из набора $[p_1..p_m]$, то он может указать набор интересных ему идентификаторов сенсоров p_i, передав строку идентификаторов сенсоров, разделенных запятыми, в параметре inPP (например, как «p3 p4 p5 p6 p7»).

Для каждой $Record_{in}$ актор хранит и считывает предыдущее состояние, которое представлено как $Record_{state}$ и представлено следующим образом:

$$Record_{state} = (topic_{1_{state}}..topic_{n_{state}}) \quad (26)$$

Компоненты каждой $topic_{i_{state}} \in Record_{state}$ представляют собой предыдущее состояние компонентов $topic_{i_{in}} \in Record_{in}$.

Выходной записью актора DetectStateChange является $Record_{out}$, которая может быть представлена следующим образом:

$$Record_{out} = \left(topic_{1_{out}}..topic_{n_{out}}\right), \quad (27)$$

где:

$$topic_{i_{out}} = (index, rcvTime, [(edge_1, outTs_1)..(edge_m, outTs_{\mathrm{m}})]). \quad (28)$$

Для каждой полученной записи $Record_{in}$, актор DetectStateChange сравнивает каждое значение сенсора $p_i \in topic_{i_{in}}$ с предыдущим состоянием сенсора $p_i \in topic_{i_{state}}$. Если они различны, то новое значение $p_i \in topic_{i_{in}}$ будет записано в $edge_i \in topic_{i_{out}}$, а $ts_i \in topic_{i_{in}}$ будет записано в $outs_i \in topic_{i_{out}}$. Данная процедура повторяется для всех тем $topic_{i_{in}} \in Record_{in}$,

Листинг 3. Псевдокод актора DetectStateChange

```
1: procedure DetectStateChange

                                              ▷ Стадия инициализации
2: if environmentVariables = null
3:       PARAMS ← GUI
4: else
5:      PARAMS ← environmentVariables
6: recordStore ←InitializeStore()

                                ▷ Цикл обработки входящих записей токенов
7: recordIn ←RecieveRecordFromInputPort()
8: recordOut ←InitializeRecord()
9: if recordStore.IsEmpty()
10:    recordStore ← recordIn
11:    goto 7
12: for each inTopicName,inTopicValue in recordIn:
13:    if inTopicName ∈ PARAMS.inTopic:
14:       for i = 1 to inTopicValue.p.Length():
15:          if inTopicValue.p[i].Name()∈ PARAMS.inPP and
             recordStore[inTopicName].p[i] <> inTopicValue.p[i]:
16:             outTopic ← recordOut[inTopicName]
17:             if outTopic.IsEmpty():
18:                outTopic.index ← inTopicValue.index
19:                outTopic.rcvTime ← inTopicValue.rcvTime
20:             outTopic.edge[i] ← inTopicValue.p[i]
21:             outTopic.outTs[i] ← inTopicValue.ts[i]
22:             recordOut[inTopicName] ← outTopic
23: if not recordOut.IsEmpty()
24:    recordStore ← recordIn
       SendRecordToOutputPort(recordOut)
25: goto 7
26: end procedure
```

или для тех тем, которые указаны пользователем в поле inTopic и в то же время находятся в $Record_{in}$. После проверки всех изменений состояния, $index \in topic_{i_{in}}$ и $rcvTime \in topic_{i_{in}}$ также будут записаны в $topic_{i_{out}}$.

После проверки необходимых изменений состояния в $Record_{in}$, актор обновляет значения $Record_{state}$ до значений в $Record_{in}$, отправляет $Record_{out}$ в выходной порт, и производит потребление следующей записи $Record_{in+1}$. Листинг 3 представляет собой псевдокод процесса работы актора DetectStateChange.

3.1.4. Актор CorrelateStateChange

Актор ***CorrelateStateChange*** решает задачу обнаружения связей между изменениями, которые произошли в одной серии данных с изменениями, которые произошли в другой серии данных путем анализа корреляции между ними. Рассмотрим список параметров данного актора:

- inTopic («Topic» в Kepler GUI): строка, содержащая перечень идентификаторов тем Kafka, в рамках которых производится отслеживание состояния.
- xxEdges («xx Edges to compare» в Kepler GUI): строка, содержащая идентификаторы датчиков, в которых отслеживаются изменения состояния (принимаются от актора DetectStateChange). Они соотносятся с изменениеми состояния датчиков, имена полей которых указаны в параметре yyEdges. Соотнесение происходит в соответствии с порядком, в котором указаны поля в данной строке.
- yyEdges («yy Edges to compare» в Kepler GUI): строка, содержащая идентификаторы датчиков, изменения состояния в которых необходимо соотнести с изменениями состояния датчиков, указанных в xxEdges.
- xxTS («xx Timestamp» в Kepler GUI): строка, содержащая идентификаторы полей временных меток для датчиков, указанных в xxEdges.
- yyTS («yy Timestamp» в Kepler GUI): строка, содержащая идентификаторы полей временных меток для датчиков, указанных в yyEdges.
- outDt («Output DT» в Kepler GUI): строка, содержащая идентификаторы полей, куда заносятся результаты рассчета корреляции состояния датчиков. В данные поля заносятся результаты сопоставления изменений, произошедших в состояниях датчиков из

xxEdges и их пар в yyEdges посредством вычисления разницы во времени между возникновением изменений состояния.

– outTS («Output timestamps» в Kepler GUI): идентификаторы полей, в которые записывается временная метка о корреляции состояния изменения (outDt).

Суть решаемой задачи сводится к расчету временной разницы между повышенным (1) и пониженным (0) состояниями каждой из входных пар сенсоров. Предположим, что p_1 — это серия данных, отображающая состояние одного из сенсоров, и в какой-то момент времени произошло изменение состояния данного сенсора (например, с «0» на «1»). Необходимо рассчитать, в каком из других анализируемых потоков данных произошло аналогичное событие (изменение с «0» на «1»), с учетом ограничений временного среза после или до момента данного события в p_1. Этот вид корреляционного анализа помогает изучить неявные взаимосвязи между сериями данных. Он также может быть использован для мониторинга и выявления закономерностей между событиями, происходящими, например, в некоторых промышленных средах.

В каждую единицу времени актор CorrelateStateChange в качестве входных данных получает запись токенов $Record_{in}$, определенную в (27). Для каждой $Record_{in}$ актор CorrelateStateChange также считывает предыдущее состояние $Record_{state}$, которое представляется следующим образом:

$$Record_{state} = \left(topic_{state_1}..topic_{state_n}\right). \quad (29)$$

Отличительной особенностью хранения $topic_{state_i}$ является то, что для осуществления анализа хранятся последние события, как связанные с переходом сенсоров из состояния «1» в состояние «0» ($topic_{state(0)_i}$) и наоборот, из состояния «0» в состояние «1» ($topic_{state(1)_i}$):

$$topic_{state_i} = \left(topic_{state(0)_i}, topic_{state(1)_i}\right), \quad (30)$$

где:

$$topic_{state(0)_i} = \left(edge_{state(0)_{i_1}}, ts_{state(0)_{i_1}}\right) .. \left(edge_{state(0)_{i_m}}, ts_{state(0)_{i_m}}\right); \quad (31)$$

$$topic_{state(1)_i} = (edge_{state(1)_{i_1}}, ts_{state(1)_{i_1}}) .. (edge_{state(1)_{i_m}}, ts_{state(1)_{i_m}}) . \quad (32)$$

Каждая пара $(edge_{state(0)_{i_a}}, ts_{state(0)_{i_a}})$ хранит отметку времени $ts_{state(0)_{i_a}}$ когда сенсор a перешел из состояния «1» в состояние «0». Аналогично, $(edge_{state(1)_{i_a}}, ts_{state(1)_{i_a}})$ хранит отметку времени $ts_{state(1)_{i_a}}$ когда сенсор a перешел из состояния «0» в состояние «1».

Результатом работы актора является запись токена $Record_{out}$, которая представлена следующим образом:

$$Record_{out} = (topic_{out_1} .. topic_{out_n}). \quad (33)$$

Каждая тема $topic_{out_i}$ представляет собой результат корреляционного анализа:

$$topic_{out_i} = \left(topic_{out(0)_i}, topic_{out(1)_i}\right), \quad (34)$$

где:

$$topic_{out(0)_i} = (DT_{out(0)_{i_1}}, ts_{out(0)_{i_1}}) .. (DT_{out(0)_{i_k}}, ts_{out(0)_{i_k}}) ; \quad (35)$$

$$topic_{out(1)_i} = (DT_{out(1)_{i_1}}, ts_{out(1)_{i_1}}) .. (DT_{out(1)_{i_k}}, ts_{out(1)_{i_k}}). \quad (36)$$

В результирующем сообщении, в каждой паре $(DT_{out(0)_{i_c}}, ts_{out(0)_{i_c}})$, $DT_{out(0)_{i_c}}$ хранит разницу во времени между текущим полученным состоянием $edges_{in_{i_a}}$, когда оно приняло значение «0», и последним обновленным $edges_{state(0)_{i_b}}$; $ts_{out(0)_{i_c}}$ хранит время последнего обновления состояния. Аналогично, при обнаружении корреляции при значении состояния равного единице, будет сформировано выходное сообщение в виде пары

Листинг 4. Псевдокод актора CorrelateStateChange

```
1: procedure CorrelateStateChange

                                              ▷ Стадия инициализации
2: if environmentVariables = null
3:       PARAMS ← GUI
4: else
5:      PARAMS ← environmentVariables
6: rStore ←InitializeCorrelateStore()

                                  ▷ Цикл обработки входящих записей токенов
7: rIn ←RecieveRecordFromInputPort()
8: rOut ←InitializeRecord()
9: for each topic in rIn:
10:    if topic.Name()∈ PARAMS.inTopic:
11:       for each edge, ts in topic:
12:          edgeIndex ← PARAMS.xxEdges.Index(edge.Name())
13:          if edgeIndex <> null:
14:             yyEdgeName ← PARAMS.yyEdges[edgeIndex]
15:             if rStore[topic.Name()][yyEdgeName][edge] = null:
16:                continue
17:             dtName ← PARAMS.outDT[edgeIndex]
18:             tsName ← PARAMS.outTS[edgeIndex]
19:             cRec ← rStore[topic.Name()][yyEdgeName][edge]
20:             rOut[topic.Name()][dtName]← xxTs-cRec.ts
21:             rOut[topic.Name()][tsName]← cRec.ts
22:             SendRecordToOutputPort(rOut)
23:       for each edge, ts in topic:
24:          edgeIndex ← PARAMS.yyEdges.Index(edge.Name())
25:          if edgeIndex <> null:
26:             yyEdgeName ← PARAMS.yyEdges[edgeIndex]
27:             rStore[topic.Name()][yyEdgeName][edge].ts ← ts
28: goto 7
29: end procedure
```

$(DT_{out(1)_{i_c}}, ts_{out(1)_{i_c}})$, где $DT_{out(1)_{i_c}}$ равен разнице во времени между текущим полученным $edges_{in_{i_a}}$, когда оно равно «1», и последним обновленным $edges_{state(1)_{i_b}}$; $ts_{out(1)_{i_c}}$ хранит время последнего обновления состояния. Листинг 4 представляет собой псевдокод корреляционного анализа, реализованного в акторе CorrelatetStateChange.

3.1.5. Актор XYState

Актор ***XYState*** обеспечивает подготовку значений на основе входящего потока данных, в формате, требуемом XYPlotter (стандартный актор плоттера в Kepler). Результатом работы актора является формирование пар значений (X, Y) на выходном порту. В качестве входных данных XYState получает запись токена, где каждая запись представлена следующим образом:

$$Record_{in} = \left(data_{in_1}, ts_{in_1}\right)..\left(data_{in_n}, ts_{in_n}\right) \quad (37)$$

где в каждой паре $(data_{in_i}, ts_{in_i})$, $data_{in_i}$ представляет собой числовое поле данных, а ts_{in_i} представляет собой метку времени, в которой данные $data_{in_i}$ генерируются или обрабатываются. Параметры актора позволяют пользователю выбрать, какие поля $data_{in_i}$ и ts_{in_i} из $Record_{in}$ должны быть разложены на два отдельных поля, X_{out_i} и Y_{out_i} соответственно. Актор обеспечивает их считывание и преобразование каждое из них в токен типа данных double.

3.2. Разработанные на базе системы Kepler микропотоки работ

На основе акторов, разработка которых была представлена в предыдущем разделе, был также разработан набор потоков работ для системы Kepler. В качестве типовой задачи обработки потоков данных в реальном времени с использованием географически-распределенных вычислительных систем используется реальный набор данных, предоставленный DEBS 2012 Grand Challenge [153]. В рамках данного набора представлен массив данных, собранных с датчиков и клапанов, установленных на промышленном оборудовании. Задержка между двумя последовательными точками данных составляет около 10 мс. Каждый элемент исходных данных представляет собой сообщение в формате Apache Avro, состоящее из 66 полей. В полях сообщения

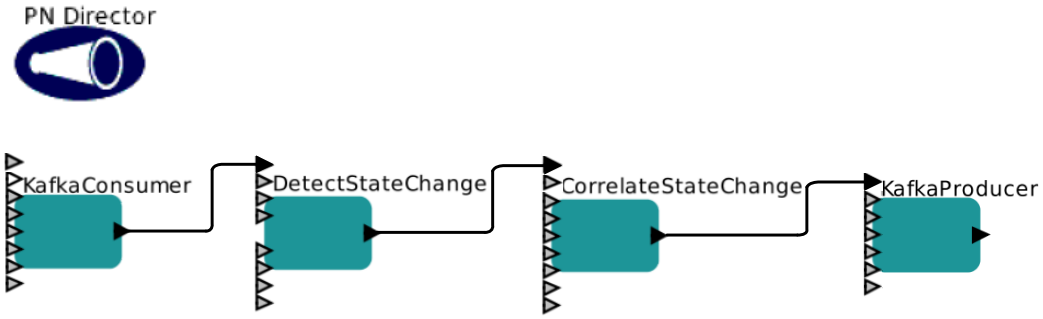


Рис. 25. Идентификация изменения состояния и корреляция изменения состояния в одном потоке работ.

содержится информация о состоянии соответствующих датчиков в момент считывания информации.

Процесс обработки данных, необходимых для решения данной задачи состоит из двух этапов. На первом этапе производится идентификация изменения состояния во входных полях между последовательными точками исходных данных и генерация сообщений о изменении состояний вместе с временными метками данного события. На втором этапе решается задача корреляции между изменением состояния датчиков и изменением состояния клапанов, а также рассчитывается временное расстояние между наступлением изменения состояния. Соответственно, можно выделить два подпотока, обеспечивающих решение данной задачи в процессе обработки данных.

Первый поток работ (см. рис. 25) разработан для включения всех необходимых этапов обработки, где исходный поток данных поступает к актору KafkaConsumer, а KafkaConsumer обеспечивает десериализацию входящих сообщений и передает их виде записей токена в актор DetectStateChange. Данный актор обеспечивает обнаружение изменений во входящем потоке данных. Каждое обнаруженное изменение отправляется как запись в актор CorrelateStateChange. Данный актор выполняет корреляционный анализ пар серий данных при получении каждой записи. Результаты работы актора CorrelateStateChange представляют собой набор потоков данных, содержащих информацию о корреляции между состояниями датчиков и клапанов.

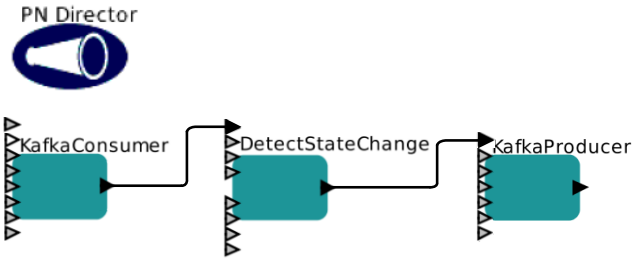


Рис. 26. Этап идентификации изменения изменения состояния в отдельном микро-потоке работ.

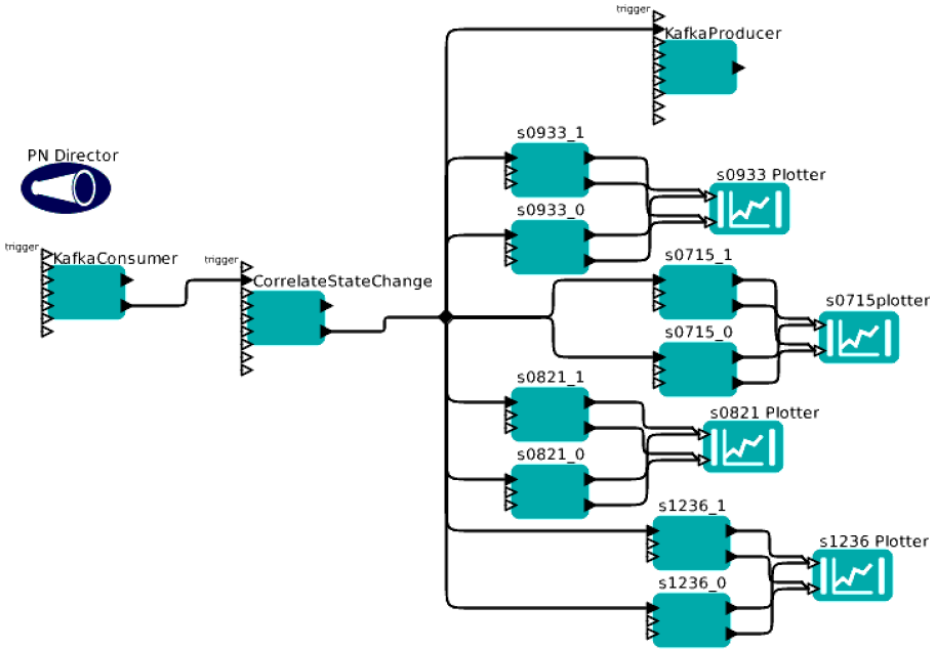


Рис. 27. Этап корреляции изменения состояния и построение результатов в отдельном микро-потоке работ.

Например, поток s_0715_0 — это поток результатов анализа корреляции между состоянием датчика с индексом 07 и клапана с индексом 15, когда был зафиксирован переход состояния в значение «0». Как только происходит обнаружение корреляции между входящими сериями данных, результаты обработки отправляются в актор KafkaProducer, который обеспечивает сериализацию конечных результатов и отправляет их в виде потока данных на платформу Kafka.

Второй поток работ (см. рис. 26) представляет собой микро-поток работ, реализующий первый этап обработки данных. В этом потоке работ исходный поток данных с сенсоров поступает на актор KafkaConsumer. KafkaConsumer обеспечивает десериализацию каждого входящего

сообщения и его передачу в виде записи токена в актор DetectStateChange. Каждое обнаруженное изменение серии данных отправляется в виде записи на актор KafkaProducer, который сериализует результаты и отправляет их в виде потока данных в очередь Kafka.

Третий поток работ (см. рис. 27) представляет собой микро-поток работ, реализующий второй этап обработки данных. На вход к актору KafkaConsumer поступает поток данных о выявленных изменениях. Актор KafkaConsumer десериализует каждое входящее сообщение и передает данные из сообщения актору CorrelateStateChange как запись токена. Актор CorrelateStateChange при получении каждой записи выполняет корреляционный анализ входящих данных на различных сериях данных.

После обнаружения корреляции актор CorrelateStateChange отправляет данные в актор KafkaProducer, который в свою очередь сериализует результаты и отправит их в Kafka. Данный микро-поток работ включает в себя также блок визуализации результатов анализа, где потоки результатов актора CorrelateStateChange разделяются и направляются в свой актор XYState для визуализации каждого потока в акторе XYPlotter.

3.3. Контейнеризация и параметризация микро-потоков работ

Перед выполнением микро-потока работ необходимо настроить его параметры, включая информацию, необходимую для связи с внешним миром. Микро-потоку работ должна быть предоставлена информация о местонахождении адреса конечной точки платформы потоков данных (в нашем случае, Apache Kafka), местонахождении хранилищ схем сообщений и идентификаторах хранилищ потоков данных для чтения и записи сообщений. Были реализованы следующие варианты параметризации микро-потоков работ:

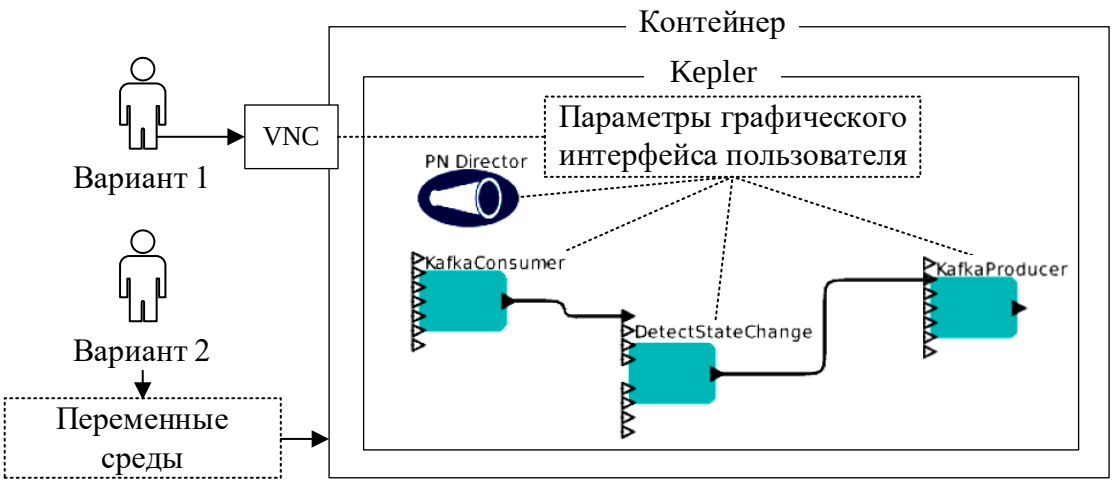


Рис. 28. Два реализованных варианта запуска микро-потока работ на базе системы управления потоками работ Kepler.

– использование удаленного доступа к графическому интерфейсу каждого потока работ и ручной ввод параметров выполнения в поток работ;
– передача параметров исполнения микро-потока работ в виде переменных среды при инициализации его контейнера. В этом случае, выполнение микро-потока работ автоматически начинается с новыми параметрами без необходимости ручного доступа к графическому интерфейсу пользователя.

Рис. 28 схематично представляет два варианта организации запуска микро-потока работ, описанных выше. Для поддержки эффективной переносимости и возможности прозрачного управления микро-потоками в туманных и облачных вычислительных средах, микро-потоки работ были помещены в контейнеры Docker. Чтобы запустить контейнер, необходимо передать контейнеру переменные среды, которые представляют параметры микро-потока работ. Ниже приведены примеры переменных среды, которые необходимо передать контейнеру в зависимости от типа микро-потока, который должен быть запущен в контейнере:

– MWF: название MWF;

```
docker run -it -d -p $KAFKAPORT -p $SCHEMAPORT \
        -e MWF=<?> -e KAFKASERVER=<?> \
        -e SCHEMAREGISTRY=<?> -e TOPICSOURCE=<?> \
        -e TOPICDESTINATION=<?> <developed_docker_image>
```

Рис. 29. Пример схемы команд Docker, используемой для запуска контейнера микро-потока работ.

– KAFKASERVER: ip-адрес сервера Kafka;
– KAFKAPORT: порт конечной точки сервера Kafka;
– SCHEMAREGISTRY: ip-адрес сервера регистрации схем;
– SCHEMAPORT: порт конечной точки сервера регистрации схем;
– TOPICSOURCE: исходная тема Kafka;
– TOPICDESTINATION: целевая тема Kafka.

Рис. 29 показывает схему команды docker run, используемую для запуска контейнера микро-потока работы на одном из примеров.

3.4. Реализованные программные утилиты

В рамках проведенного исследования был также разработан набор программных утилит и библиотек, поддерживающих проведение экспериментов. В данный набор входят компоненты, обеспечивающие генерацию тестовых потоков данных систем интернета вещей, синхронизацию данных между географически-распределенными развертываниями платформы Apache Kafka, а также автоматизацию разбиения монолитных потоков работ на микро-потоки работ.

3.4.1. Симулятор датчиков

При проведении вычислительных экспериментов разработанных программных компонентов поддержки микро-потоков работ были использованы данные реальных датчиков из промышленной среды. Наборы таких данных обычно предоставляются в виде текстовых файлов или баз данных. Для того,

Листинг 5. Псевдокод симулятора датчика

```
1: procedure IoTSimulator

                                          ▷ Стадия инициализации
2: PARAMS ← environmentVariables
3: con ← ConnectToKafkaProduce(PARAMS.KAFKA_SERVER,
   PARAMS.TOPIC)
4: schema ← PARAMS.OUTSCHEMA
5: SendSchema(PARAMS.SCHEMA_REG_SERVER, schema)
6: dataset ← OpenRead( PARAMS.DATASET)

                              ▷ Цикл обработки входящих записей токенов
7: if GetCurrentTime()≥ PARAMS.STOPTIME or EOF(dataset)
8:    EXIT
9: record ←GetNextRecord(dataset)
10: message ←Initialize(schema)
11: for each fieldName, fieldValue in record:
12:    message.Insert(fieldName, fieldValue)
13: ProduceMessage(con, message)
14: Wait(PARAMS.INTERVAL)
15: goto 7
16: end procedure
```

чтобы протестировать возможность обработки потоков данных микро-потоками работ, был разработан *симулятор датчиков*, который формирует поток записей из набора данных и передает их в очередь сообщений Apache Kafka.

Листинг 5 представляет собой псевдокод реализованного симулятора датчиков. Рассмотрим список параметров, передаваемых при запуске в симулятор датчиков:

– DATASET: строка, содержащая путь к используемому набору данных.
– KAFKA_SERVER: строка, содержащая ip-адрес и порт сервера очереди сообщений Apache Kafka.
– TOPIC: строка, содержащая имя темы Kafka, в которую производится отправка потока данных.
– SCHEMA_REG_SERVER: строка, содержащая ip-адрес и порт сервера регистрации схемы данных Apache Avro.

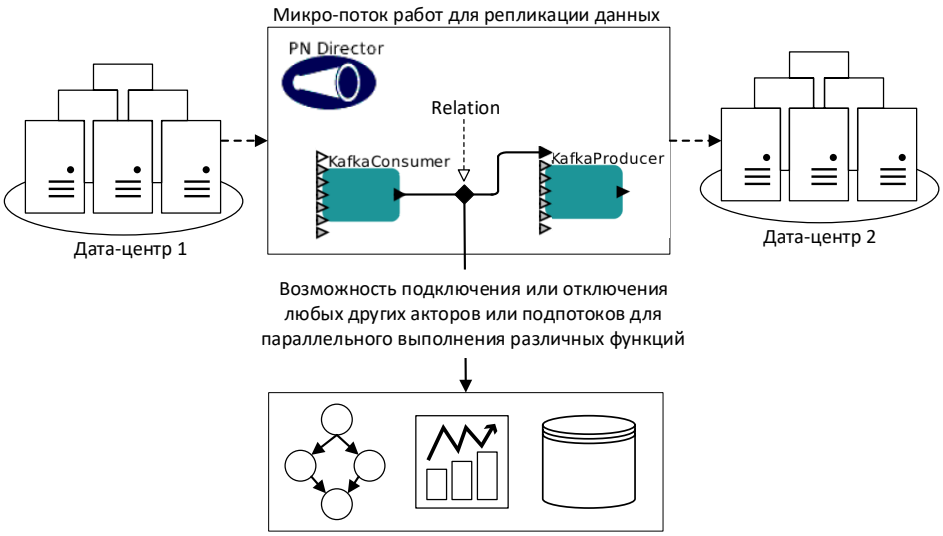


Рис. 30. Микро-поток работ для репликации данных.

– OUTSCHEMA: пецификаци схемы передаваемых данных в формате Apache Avro.
– STOPTIME: метка времени, до которого должна происходить генерация данных.
– INTERVAL: интервал между предачей последовательных сообщений в потоке данных.

3.4.2. Репликатор данных

Для организации синхронизации данных между разделенными центрами обработки данных в туманных средах был разработан механизм репликации данных между двумя или более географически разделенными кластерами Apache Kafka. Механизм репликации разработан на основе концепции микро-потоков работ с использованием акторов KafkaProducer и KafkaConsumer на базе системы Kepler. Рис. 29 представляет собой иллюстрацию процесса репликации данных с использованием разработанного репликатора. Репликатор позволяет реплицировать темы с одного кластера Kafka на другой, где актор KafkaConsumer получает данные из темы Kafka и передает их в компонент *Relation*. Компонент Relation (который также можно назвать коннектором) — это собственный компонент Kepler, который позволяет

Листинг 6. Рефакторинг микро-потоков работ

```
 1: procedure Refactor
 2: wf, subWfs ← ReadWorkflow(PARAMS.SOURCE)
 3: Z ←InitializeZ(wf, subWfs)    ▷ Формула (16)
 4: Z ←IdentifyInternalExternalEdges(Z)    ▷ Формула (17)
 5: for i from 1 to subWfs.Length():
 6:     MWF[i]←InitializeM(subWfs[i],Z)    ▷ Формула (20)
 7:     MWF[i]←InternalEdges(MWF[i],Z)    ▷ Формула (21)
 8:     MWF[i]←AddCv(MWF[i],Z)    ▷ Формула (22)
 9:     MWF[i]←AddPv(MWF[i],Z)    ▷ Формула (23)
10: WriteMWFs(MWF)
11: end procedure
```

акторам отправлять выходные данные нескольким компонентам Kepler одновременно. После этого актор KafkaProducer получает входные данные от Relation и отправляет потоки в другой кластер Kafka.

Микро-поток работ, обеспечивающий работу репликатора, позволяет добавлять дополнительные возможности по анализу и логированию процесса репликации путем подключения к Relation акторов и компонентов Kepler. Например, доступна возможность визуализации процесса репликации с помощью актора плоттера Kepler, архивирование потока в базу данных с помощью актора базы данных Kepler и так далее. Следует отметить, что все эти задачи выполняются параллельно, поскольку репликатор микро-потока работает с PN директором.

3.4.3. Утилита рефакторинга монолитных потоков работ

Для реализации и испытания алгоритма рефакторинга монолитного потока работ на набор независимых микро-потоков работ, представленного в разделе 2.3, было разработано приложение. Листинг 6 представляет собой псевдокод разработанной утилиты. На вход подается описание монолитного потока работ вместе с наборами вершин, формирующих под-потоки работ. Результатом выполнения утилиты является набор файлов, каждый из которых

представляет описание отдельного микро-потока работ, сформированного в результате рефакторинга.

3.5. Вывод по главе 3

В данной главе описана реализация акторов системы Kepler, обеспечивающих взаимодействие с системой поточной обработки данных Apache Kafka, системой хранения схем данных Apache Avro, а также поддерживающие выполнение типовых задач обработки данных в поточном режиме. Для тестирования подхода микро-потоков работ, на основе разработанных акторов, были спроектированы и реализованы потоки работ, обеспечивающие обработку данных в поточном режиме. В качестве типовой задачи обработки данных была взята задача DEBS 2012 Grand Challenge, в рамках которой производится анализ данных, собираемых в режиме реального времени с датчиков и клапанов, установленных на промышленном оборудовании. Для поддержки проведения вычислительных экспериментов также был реализован ряд утилит, обеспечивающих эмуляцию данных устройств IoT, репликацию данных между географически-распределенными серверами Apache Kafka, а также рефакторинг монолитных потоков работ. Основные результаты, описанные в этой главе, были опубликованы в [6,8,85,124]. Основные исходные коды третьей главы свободно доступны в сети Интернет: https://github.com/alaasamameer/Micro-Workflows.

ГЛАВА 4. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

В данной главе представлены результаты тестирования модели микро-потоков работ. Вычислительные эксперименты проводились с использованием акторов и потоков работ, представленных в главе 3. Ключевой задачей вычислительных экспериментов является оценка применимости модели и алгоритмов микро-потоков работ для организации обработки потоков данных, в том числе данных IoT устройств, в распределенных вычислительных средах. Критериями для оценки эффективности данной модели выбраны метрики, отображающие среднюю задержку обработки данных, объем данных, передаваемых через глобальные сети для обеспечения обработки данных, среднее время доставки сообщений. Вычислительные эксперименты включают в себя реализацию и тестирование алгоритма по рефакторингу монолитного потока работ на микро-потоки работ; реализацию обработки данных с использованием локальных и удаленных вычислительных ресурсов; сопоставлению применения монолитных и микро-потоков работ для обработки потоков данных; а также оценку возможностей по реализации Stateful вычислений в контексте экстренной остановки и переноса вычислительного процесса в микро-потоке работ на другой вычислительный узел. При проведении вычислительных экспериментов использовались вычислительные ресурсы управления информатизации ЮУрГУ, ресурсы Лаборатории суперкомпьютерного моделирования ЮУрГУ, а также облачные ресурсы платформы Yandex.Cloud.

4.1. Эксперимент по рефакторингу монолитного потока работ

Для тестирования алгоритма рефакторинга монолитного потока работ на набор микро-потоков работ, представленного в разделе 2.3, была разработана утилита рефакторинга монолитных потоков работ (см. 3.4.3). В качестве

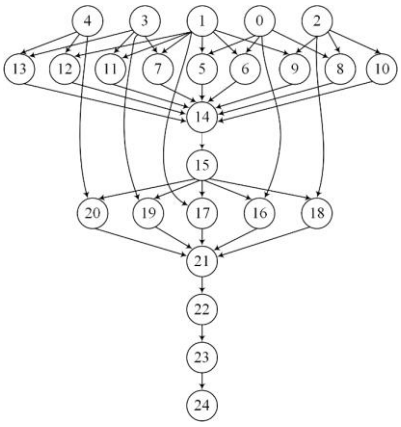


Рис. 31. Типовой поток работ «Montage 25».

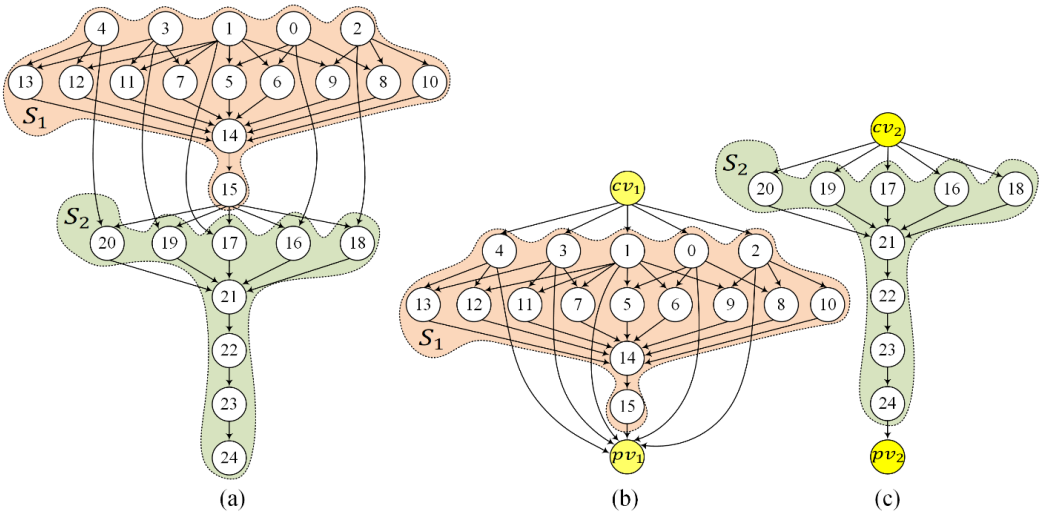


Рис 32. Рефакторинг «Montage 25» на два микро-потока работ.

монолитного потока работ, на основании которого производилось тестирование предложенного алгоритма, был взят поток работ «Montage 25» (см. рис. 31) [60].

Процесс рефакторинга потока работ «Montage 25» на набор микро-потоков работ начинается с ручного разделения входного потока работ на подпотоки, путем указания вершин потока работ входящих в каждый из подпотоков. На основания множества вершин в каждом из подпотоков, а также наборов ребер из начального монолитного потока работ производится его рефакторинг в набор микро-потоков работ. Эксперимент проводится на персональном компьютере, оснащенном процессором Intel(R) Core (TM) i5-4210U и 8 ГБ оперативной памяти.

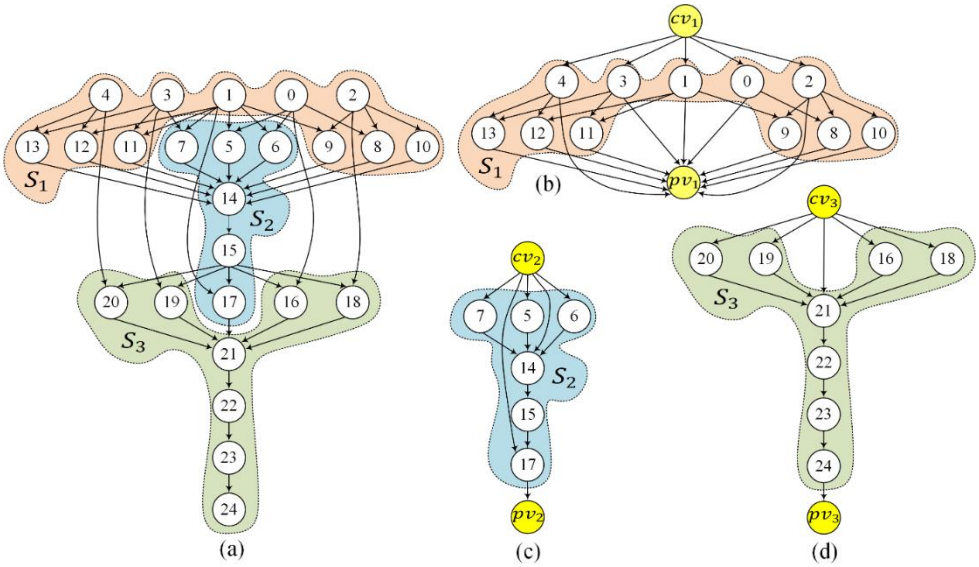


Рис. 33. Рефакторинг «Montage 25» на три микро-потока работ.

В рамках первого испытания поток работ «Montage 25» был разделен на два микро-потока работ. На рис 32a представлена визуализация разделения «Montage 25» на два подпотока: S_1 и S_2. На рис 32b и рис 32c показана визуализация результатов выполнения алгоритма рефакторинга над потоком работ «Montage 25» для выделения двух микро-потоков работ.

В рамках второго эксперимента тот же поток работ «Montage 25» был разделен на три микро-потока работ. На рис. 33a представлена визуализация разделения этого потока работ на три подпотока S_1, S_2, и S_3. На рис. 33b, рис. 33c и рис. 33d представлена визуализация результатов рефакторинга потока работ «Montage 25» на три микро-потока работ.

4.2. Эксперимент по сравнению монолитного потока работ и микро-потока работ для обработки потоков данных в реальном времени

Этот эксперимент показывает преимущество разработанной концепции микро-потоков работ по сравнению с существующим подходом к организации вычислительного процесса в виде монолитного потока работ (см. 2.1) при обработке потоков данных в реальном времени. Как было отмечено в

разделе 1.4.3, классические научные потоки работ реализуют концепцию пакетной обработки данных, что означает необходимость предварительной подготовки массива данных для обработки. В отличие от монолитного подхода, подход микро-потоков работ поддерживает возможность потоковой обработки данных. Критерием оценки в рамках данного эксперимента является среднее время реакции на событие, информация о котором может быть извлечена из потока данных в реальном времени.

В рамках данного эксперимента моделируется типовое поведение монолитных и микро-потоков работ. Вычислительный процесс монолитного потока работ организован в виде пакетной обработки данных. Для запуска обработки данных в пакетном режиме, необходимо провести их предварительный сбор и размещение на сервере хранения, с которого они будут потребляться начальным узлом монолитного потока работ. После того, как произведено планирование вычислительных задач, составляющих монолитный поток работ, и они распределены по вычислительным узлам вычислительной системы, запускается считывание данных и их пакетная обработка. Поток работ завершает свое выполнение после завершения выполнения последней задачи и освобождения занятых системных ресурсов. Чтобы запустить монолитный поток работ для нового пакета данных, все вышеперечисленные задачи должны быть выполнены заново. Так как пакет данных должен быть заранее подготовлен для обработки монолитным потоком работ, необходимо использование специальных стратегий распределения данных между вычислительными ресурсами. Этот этап перераспределения данных также оказывает значительное влияние на общее время выполнения потока работ. С другой стороны, при обработке каждой новой порции потока данных необходимо прилагать дополнительные вычислительные усилия для преобразования информации из потока данных (набор независимых сообщений) в пакет данных (содержащий полный набор данных за определенный период). В данном эксперименте сборщик потока данных разработан как предварительный

шаг в этапе сбора данных. Он реконфигурируется для сбора всех сообщений, содержащих информацию, за заданный период времени. Сборщик данных обеспечивает сериализацию собранных данных в виде пакета данных (выходного файла), содержащего все данные за требуемый период и затем запускает монолитный поток данных для обработки этого пакета. Результаты обработки пакета данных генерируются по окончанию выполнения последней задачи в указанном монолитном потоке работ.

В рамках микро-потока работ организован процесс поточной обработки данных в режиме постоянного выполнения, без заранее определенного времени завершения обработки данных. Такой подход позволяет избежать задержек, характерных для пакетной обработки данных, связанных с необходимостью предварительного накопления пакета данных, планирования выполнения потока данных на вычислительных узлах, накладных расходах на развертывание и запуск процессов обработки данных и др. Также, обработка потока данных позволяет обеспечить на порядки более низкое время реакции системы на события, информация о которых может быть идентифицирована в потоке данных.

Эксперимент реализован на одном узле, оснащенном процессором Intel(R) Core (TM) i5-4210U и 8 ГБ оперативной памяти. Производится одновременный запуск симулятора датчиков (см. раздел 3.4.1), микро-потока работ и монолитного потока работ, обеспечивающих обработку потока данных, получаемых с этих датчиков. После завершения генерации данных производится ожидание получения информации о событиях, идентифицированных микро-потоком работ и монолитным потоком работ. Затем производится сравнение времени отклика и значений полученных результатов обработки данных. В рамках эксперимента было произведено два испытания. В рамках *испытания 1* реализуется обработка потока данных в течение 30 минут. В рамках *испытания 2* реализуется обработка потока данных в течение 1 часа.

В каждом испытании симулятор датчиков формирует поток данных, где каждое сообщение потока данных отражает информацию с 8 датчиков в момент генерации этого сообщения. Задача обработки данных состоит в том, чтобы обеспечить отслеживание в изменении значений датчиков, и зафиксировать время этого изменения. Изменение состояния датчиков может произойти в любой момент времени в течение тестового периода. Задержка между генерируемыми сообщениями составляет порядка 9,5 миллисекунд.

В рамках испытания 1, в течение 30 минут была произведена обработка 18 000 сообщений потока данных. При этом как монолитный, так и микро-поток работ идентифицировали 150 событий, связанных с изменением состояния датчиков. В рамках испытания 2, в течение 60 минут была произведена обработка 378 948 сообщений потока данных. При этом как монолитный, так и микро-поток работ идентифицировали 383 событий, связанных с изменением состояния датчиков. Результаты, полученные в ходе испытаний, представлены в табл. 2 и табл. 3. В качестве метрик для оценки результатов испытаний были использованы следующие показатели:

- **время ответа**: среднее время ответа на событие в потоке данных, которое рассчитывается как среднее значение для разницы между моментом генерации события источником данных и моментом обнаружения этого события при его обработке;
- **объем выходных данных:** средний объем входных данных, необходимых для начала каждого вычисления;
- **объем результирующих данных:** средний объем результирующих данных, получаемых в результате обработки каждой порции данных.

Табл. 2. Результаты испытания 1 (обработка 30 минут потока данных).

	Время ответа	Объем входных данных	Объем результирующих данных
Монолитный поток работ	670 732 мс.	27.4 Мбайт	46 Кбайт
Микро-поток работ	1.4 мс.	1 Кбайт	1 Кбайт

Табл. 3. Результаты испытания 2 (обработка 60 минут потока данных).

	Время ответа	Объем входных данных	Объем результирующих данных
Монолитный поток работ	164 285 8 мс.	57.8 Мбайт	114 Кбайт
Микро-поток работ	1.4 мс.	1 Кбайт	1 Кбайт

Результаты, полученные в этом эксперименте, позволяют сделать следующие выводы в плане обработки потоковых данных при необходимости реакции на события в режиме, близком к реальному времени:

1) При организации обработки потока данных посредством микро-потока работ среднее время ответа оказалось значительно меньше (1.4 мс.), чем время ответа, обеспечиваемое монолитным потоком работ (11.1 мин. и 27.3 мин. в испытании 1 и 2 соответственно). Это достигается за счет того, что микро-поток работ получает данные и обрабатывает данные в потоковом режиме, обеспечивая предоставление результатов, как только они становятся доступными, в то время как монолитный поток ждет, пока завершится этап подготовки данных для запуска их обработки.
2) Среднее время ответа на события при обработке в формате микро-потока работ не зависит от общего периода выполнения, в то время как при реализации обработки данных в пакетном режиме монолитного потока работ, среднее время предоставления ответов возрастает при увеличении периода выполнения в связи с увеличением периода подготовки данных и увеличением размера входных данных.

3) Объем данных, необходимых для инициализации вычислительного процесса в формате микро-потока работ, очень мал по сравнению с монолитным потоком работ (1 Кбайт против 27.4 Мбайт или 57.8 Мбайт в рамках испытаний 1 и 2 соответственно). Аналогично, средний размер сообщений с результатами обработки данных микро-потока работ также меньше по сравнению с монолитным потоком работ (1 Кбайт против 46 Кбайт или 114 Кбайт в рамках испытаний 1 и 2 соответственно). Это показывает, что микро-поток работ более подходит для развертывания в условиях ограниченных сетевых и вычислительных ресурсов, в том числе, на узлах туманных вычислительных систем.

4.3. Эксперимент по локальному и распределенному развертыванию микро-потоков работ

4.3.1. Исходные данные для проведения экспериментов

Основой для проведения дальнейших вычислительных экспериментов, представленных в данной главе, являются реальные данные датчиков производственного оборудования, доступные в рамках DEBS 2012 Grand Challenge [153]. Каждое сообщение в исходных данных включает в себя 66 полей, представляющих собой информацию о состоянии систем химической промышленности, полученную на основании данных из массива сенсоров, установленных на соответствующем оборудовании. Задержка между двумя последовательными сообщениями в потоке данных индустриального IoT составляет около 10 мс.

Задача обработки данных, которая рассматривается в рамках данной серии вычислительных экспериментов, включает в себя две группы операторов. Первая группа операторов обеспечивает обнаружение изменения состояния входных полей и выдает их вместе с метками о времени изменения состояния. Вторая группа операторов коррелирует изменение состояния

датчиков и изменение состояния клапанов, вычисляя разницу во времени между наступлением изменений состояния и выдает их вместе с метками времени, в которых обнаружены данные корреляции. Решение данных задач в виде потоков работ представлено в разделе 3.2.

Для моделирования отправки информации датчиками IoT, был разработан компонент-эмулятор, который считывает информацию из файла исходных данных и отправляет их в виде потока данных в систему поддержки поточной обработки данных Apache Kafka с заранее определенной величиной задержки между отправляемыми сообщениями. Реализация данного эмулятора была представлена в разделе 3.4.1. Для автоматизации развертывания Apache Kafka и реестра схем Apache Avro используются Docker-образы, предоставляемые Lenses.io[3].

4.3.2. Методика проведения эксперимента

Возможность распределения нагрузки по узлам распределенной вычислительной системы является необходимым условием для приложений, функционирующих в рамках архитектуры туманных вычислений. В связи с этим, фокус первого эксперимента был направлен на оценку возможности применения концепции микро-потоков работ для разделения сильно-связанных зависимостей в монолитных потоках работ на более мелкие и автономные микро-потоки работ, путем организации связей между ними посредством среды потоковой обработки данных, а также оценку объема накладных расходов, возникающих при распределенном развертывании микро-потоков работ.

В рамках этого эксперимента, были протестированы два варианта развертывания потоков работ. Во первом варианте развертывания

[3] https://lenses.io/

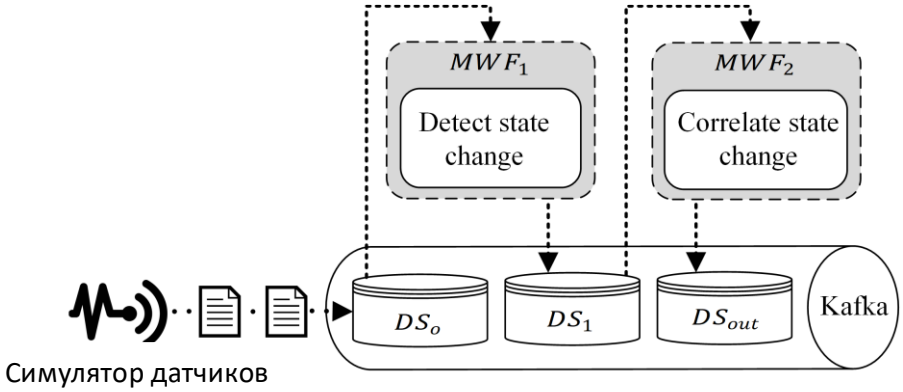


Рис. 34. Организация обработки потока данных в виде двух микро-потоков работ.

(*испытание 1)* единый поток работ был разбит на два независимых микро-потока работ, но они были развернуты на одном узле. Во втором варианте развертывания (*испытание 2*) два микро-потока работ были развернуты на отдельных вычислительных узлах, находящихся в рамках одной локальной вычислительной сети.

Микро-потоки работ были реализованы с помощью системы управления потоками работ Kepler и упакованы в контейнеры Docker. В качестве платформы потоковой передачи данных для организации обмена сообщениями между микро-потоками работы использовался Apache Kafka. Хранилища потоков данных были реализованы в виде тем Kafka (см. рис. 34).

Процесс генерации данных в рамках вычислительного эксперимента реализуется следующим образом. Симулятор датчиков получает данные из предварительно записанной базы данных, в которой хранятся данные DEBS 2012 Grand Challenge, обеспечивает сериализацию сообщений в формате, соответствующем заранее сформированной схеме Apache Avro, и передает их на сервер Apache Kafka в тему DS_0. После получения подтверждения успешного приема сообщения от сервера Apache Kafka, перед генерацией и отправкой следующего сообщения добавляется задержка в 8 мс. Суть эксперимента состоит в симуляции работы системы обработки потока данных в течение 24 часов.

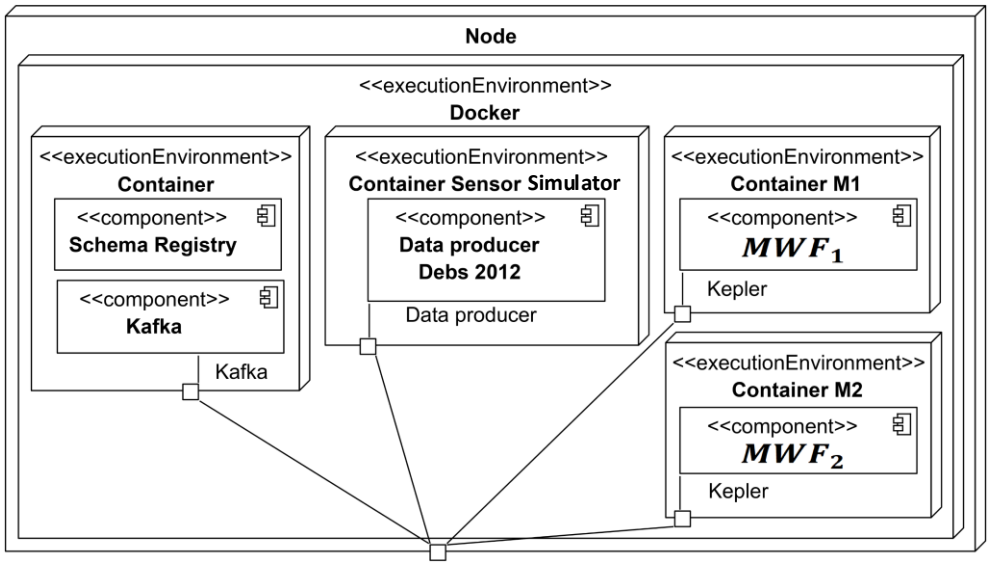


Рис. 35. Диаграмма развертывания эксперимента при испытании 1.

Для того, чтобы отслеживать метрики, связанные со временем обработки и латентностью передачи сообщений, в исходные структуры передаваемых данных были добавлены следующие дополнительные поля:

1) **mX_orgts**: метка времени, обозначающая момент, когда симулятор датчиков отправляет исходное сообщение номер X.
2) **mX_rcvts**: метка времени, обозначающая момент, когда микро-поток работ получает исходное сообщение X из исходной темы Apache Kafka.
3) **K_sentts**: метка времени, обозначающая момент, когда актер KafkaProducer отправляет итоговое сообщение в конечную тему Apache Kafka.

Рассмотрим, каким образом было организовано развертывание компонентов, отвечающих за генерацию и обработку данных в рамках данного эксперимента. В рамках испытания 1, микро-потоки работ, на которые был разбит единый поток работ, были развернуты на одном узле.

На диаграмме развертывания (см. рис. 35) видно, что в случае испытания 2 экземпляры потокового промежуточного программного обеспечения (Apache Kafka), регистра схем (Apache Avro), симулятор датчиков, микро-потоки работ MWF_1 и MWF_2 упакованы в отдельные контейнеры Docker и

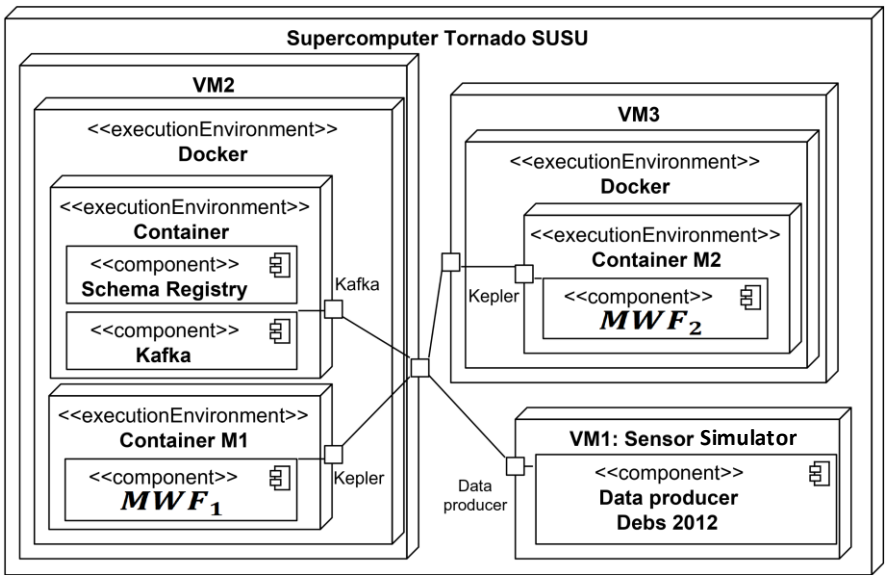


Рис. 36. Диаграмма развертывания эксперимента при испытании 2.

развернуты на одном вычислительном узле, оснащенным двухъядерным процессором Intel Core i7-4600U 2,1 ГГц с 8 ГБ оперативной памяти.

Испытание 2 было реализовано на ресурсах суперкомпьютера «Торнадо» Южно-Уральского государственного университета. Как видно из диаграммы развертывания, представленной на рис. 36, в рамках этого испытания задействовано 3 виртуальных машины. На виртуальной машине симулятора датчиков $VM1$ имитируется процесс генерации данных датчиков IoT. Он потребляет данные из базы данных DEBS 2012 и последовательно их публикует во входную тему Kafka, развернутую в контейнере в $VM2$. Микропотоки работ MWF_1 и MWF_2 упакованы в контейнеры Docker, и развернуты на виртуальных машинах $VM2$ и $VM3$. $VM1$ и $VM3$ находятся на одном физическом узле (4 ГБ оперативной памяти и 4 ядра процессора Intel Xeon X5680). $VM2$ выполняется на отдельном физическом узле (12 ГБ оперативной памяти и 8 ядер процессора Intel Xeon X5680). Связь между виртуальными машинами организована через внешний физический узел, выполняющий роль виртуального роутера, подключенного через сеть Gigabit Ethernet.

Табл. 4. Сопоставление значений метрик, полученных при проведении эксперимента

Метрики	**Испытание 1 (локальное развертывание микро-потоков работ)**	**Испытание 2 (распределенное развертывание микро-потоков работ)**
Время тестирования	24 часа	24 часа
Количество обработанных сообщений	9 180 056	7 328 844
Av_SM	9.4 мс.	11.8 мс.
Av_TAT	1.3 мс.	3.2 мс.
AV_L12	—	3.5 мс.

4.3.3. Оценка и анализ результатов эксперимента

Для оценки эффективности обмена данными и их обработки в рамках микро-потоков работ, по результатам проведенных испытаний были рассчитаны значения следующих метрик:

1) **Av_SM** (средний интервал между исходными сообщениями): средняя задержка по времени между двумя последовательными исходными сообщениями с данными, переданными симулятором датчиков на сервер Apache Kafka.
2) **Av_TAT** (среднее время обработки): средний временной интервал, требуемый для создания одного итогового сообщения микро-потоком работ. Данная метрика позволяет оценить, какое время требуется системе обработки данных, состоящей из промежуточного ПО потоковой обработки данных и сервисов микро-потоков работ на обработку данных и генерацию результирующего сообщения.
3) **Av_L12** (средняя задержка): средняя сетевая задержка между узлами, на которых развернуты микро-потоки работ MWF_1 и MWF_2.

Результаты значений метрик, полученные в рамках проведенного эксперимента, представлены в табл. 4. Разница в количестве сообщений, переданных

за время эксперимента, объясняется тем, что генератор данных был настроен на запуск 8 мс. задержки только после получения от сервера Apache Kafka подтверждения получения предыдущего сообщения. Таким образом, в случае испытания 2 на значение этого показателя влияла латентность между $VM1$, на которой был развернут эмулятор IoT датчиков, и $VM2$, на которой был развернут сервер Apache Kafka. Анализ результатов эксперимента позволяет сформулировать следующие выводы:

– Внедрение потоковой обработки данных в поток работ, а также разбиение потока работ на независимые микро-потоки позволяет обеспечить обработку данных из IoT в потоковом режиме, близком к реальному времени. Среднее время обработки данных как в испытании 1, так и в испытании 2 оказалось значительно меньше, чем период генерации датчиком исходных данных.
– Вычислительные процессы отдельных микро-потоков работ могут быть распределены по узлам вычислительной сети. При таком распределении время получения ответа увеличивается по крайней мере на величину задержки между узлом где размещен микро-поток работ и узлом, на котором расположена платформа потоковой передачи событий. Однако, результаты испытания 2 позволяют сделать вывод, что даже в этом случае время генерации ответа остается меньше скорости генерации данных с датчиков.

4.4. Группа экспериментов по обработке данных с сохранением состояния средствами микро-потоков работ

В разделе 1.3.4 было показано, что вопрос организации вычислений с сохранением состояния действительно важен в задачах обработки потоковых данных, например, при создании цифровых двойников. Развертывание сервисов

обработки данных в контейнерах существенно влияет на возможность сохранения состояния таких микросервисов. Состояние вычислительного сервиса может быть потеряно при наступлении ряда событий, таких как прекращение работы приложения в случае нехватки памяти или вычислительных ресурсов, миграции приложения в связи с ребалансировкой и т.д. (см. 1.3.5). В таких случаях после перезапуска сервиса его локальные данные будут пустыми. При обращении данного сервиса к системе обработки потоков данных, такой как Apache Kafka, будет произведен перезапуск обработки данных по последнему зафиксированному смещению в теме. Возможным решением данной проблемы является использование Kafka Streams DSL, в рамках которого поддерживается обработка данных с сохранением состояния путем создания специальных тем Apache Kafka для хранения промежуточных результатов при преобразовании данных (см. 1.3.7). Такое решение позволяет восстановить состояние приложения посредством сканирования промежуточных тем до последнего зафиксированного смещения, что обеспечивает высокий уровень отказоустойчивости.

Данная группа экспериментов посвящена оценке возможностей поддержки обработки данных с сохранением состояния с использованием концепции микро-потоков работ. В первом эксперименте была проведена оценка накладных расходов при использовании Kafka Streams DSL для реализации вычислений с сохранением состояния. Во втором эксперименте был исследован аспект возможности восстановления работы микро-потока работ после остановки или выхода из строя контейнера, без влияния на финальные результаты обработки данных.

4.4.1. Эксперимент по оценке эффективности Kafka Streams DSL для реализации вычислений с сохранением состояния

Данный эксперимент разработан для анализа времени отклика при реализации в микро-потоках работ обработки данных с сохранением состояния. В

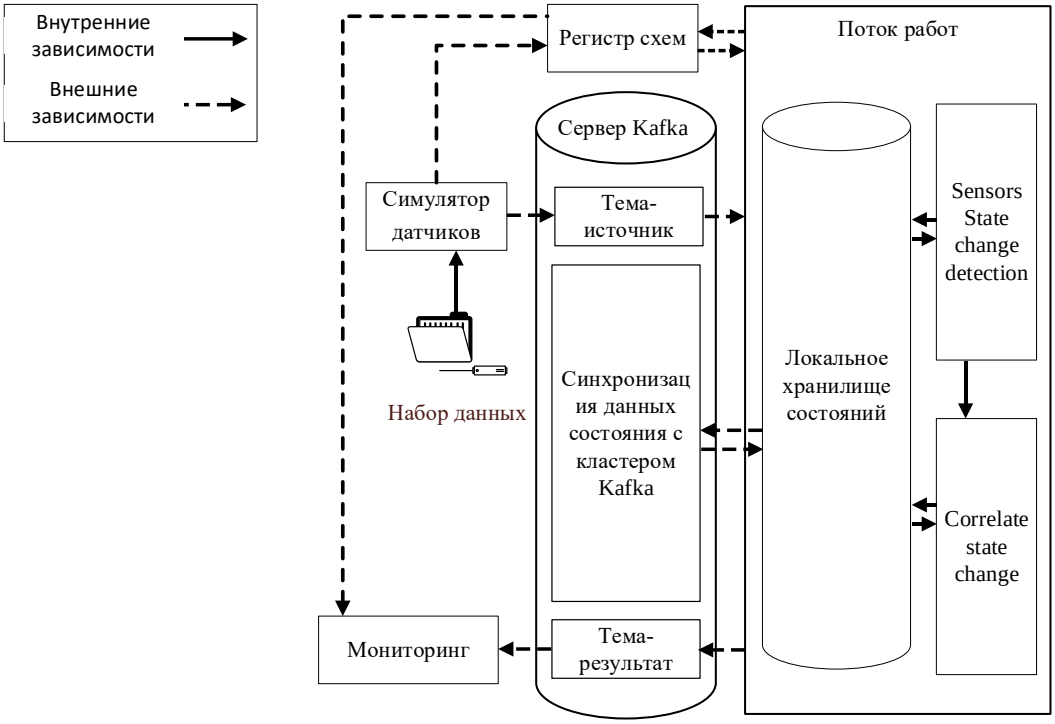


Рис. 37. Диаграмма обмена данными между компонентами системы при реализации эксперимента по оценке эффективности Kafka Streams DSL.

рамках эксперимента была произведена модификация единого потока работ, который является реализацией процесса обработки запросов над данными DEBS 2012 Grand Challenge.

В процесс обработки данных был интегрирован механизм синхронизации локального состояния путем реализации механизма поддержки вычислений с сохранением состояния в Kafka Streams DSL. На рис. 37 приведена диаграмма, отображающая процесс обмена данными между компонентами системы при реализации данного эксперимента. Компоненты эксперимента были развернуты на одном вычислительном узле со следующими характеристиками: двухъядерный процессор Intel Core i7-4600U 2.1 ГГц с 16 ГБ оперативной памяти. Результаты эксперимента приведены в табл. 5. Процесс измерения значений указанных метрик совпадает с указанным в разделе 4.3.3. За один час проведения эксперимента система обработала более 379 тысяч сообщений. При этом средняя задержка ответа при обработке сообщения составила 70.7 мс.

Табл. 5. Значения метрик, полученные в результате проведения эксперимента по оценке эффективности Kafka Streams DSL

Метрики	**Значение**
Время тестирования	1 час
Количество сообщений	379 820
Av_SM	9.5 мс
Av_TAT	70.7 мс

Сопоставление значений метрик Av_SM и Av_TAT полученных в рамках данного эксперимента (см. табл. 5) со значениями аналогичного испытания, но без реализации механизмов организации вычислений с сохранением состояния (см. табл. 4) позволяют сделать следующие выводы. При сохранении среднего интервала между исходными сообщениями, интеграция вычислений с поддержкой сохранения состояния значительно влияет на среднее время ответа (метрика Av_TAT). Значение данной метрики увеличивается с 1.3 мс. в режиме работы без сохранения состояния до 70.7 мс в режиме сохранения состояния. Это обусловлено увеличением накладных расходов на задачи, связанные с инициализацией локального хранилища состояния, организацией синхронизации состояния, как с локальным хранилищем состояний, так и с сервером Apache Kafka, находящемся в отдельном контейнере. С другой стороны, реализация механизмов сохранения состояния может быть включена/выключена отдельно для некоторых микро-потоков работ, что позволяет обеспечить более тонкую настройку системы с точки зрения ее устойчивости к сбоям и/или скорости реакции на входящие сообщения.

4.4.2. Эксперимент по живой миграции микро-потока работ

Возможность восстановления вычислительного процесса после сбоя или его преднамеренной остановки является необходимым условием при реализации промышленных систем обработки потоков данных. При реализации вычислений без сохранения состояния, такая задача не вызывает трудностей.

Отсутствие в сервисе внутреннего контекста обработки данных позволяет в любой момент времени передать вычислительную нагрузку на любой другой экземпляр данного сервиса. Однако при реализации вычислений с сохранением состояния, требуются специальные механизмы, поддерживающие сохранение и перенос состояния.

В этом эксперименте проводится испытание по оценке возможности восстановления вычислительного процесса микро-потока работ, реализующего процесс обработки данных с сохранением состояния, после остановки или выхода из строя его контейнера. Кроме того, проверяется возможность использования функции синхронизации локального хранилища состояния с промежуточными темами Apache Kafka в качестве основы для переноса вычислительной задачи в новый контейнер, сохраняя при этом непрерывность результатов из предыдущей точки остановки.

Эксперимент был развернут на узле со следующими характеристиками: двухъядерный процессор Intel Core i7-4600U 2,1 ГГц с 16 ГБ оперативной памяти. Исходные данные отправляются в исходную тему Apache Kafka. Разница во времени между двумя последовательными сообщениями исходных данных составляет порядка 10 мс. Тест начинается с инициализации двух контейнеров (ksA1, ksB) с микро-потоком работ, представленном в разделе 4.4.1. Каждый из запущенных микро-потоков работ получает уникальный идентификатор приложения и различные темы вывода в системе Apache Kafka (см. точки 1 и 2 на рис. 38). После этого производится запуск симулятора датчиков IoT, который генерирует поток данных в течение одного часа. Для симуляции экстренной остановки процесса обработки данных, через 20 минут после начала эксперимента завершается работа контейнера ksA1 (см. точку 3 на рис. 38), и запускается новый контейнер (ksA2) с тем же микро-потоком работ, той же темой ввода и вывода, и тем же

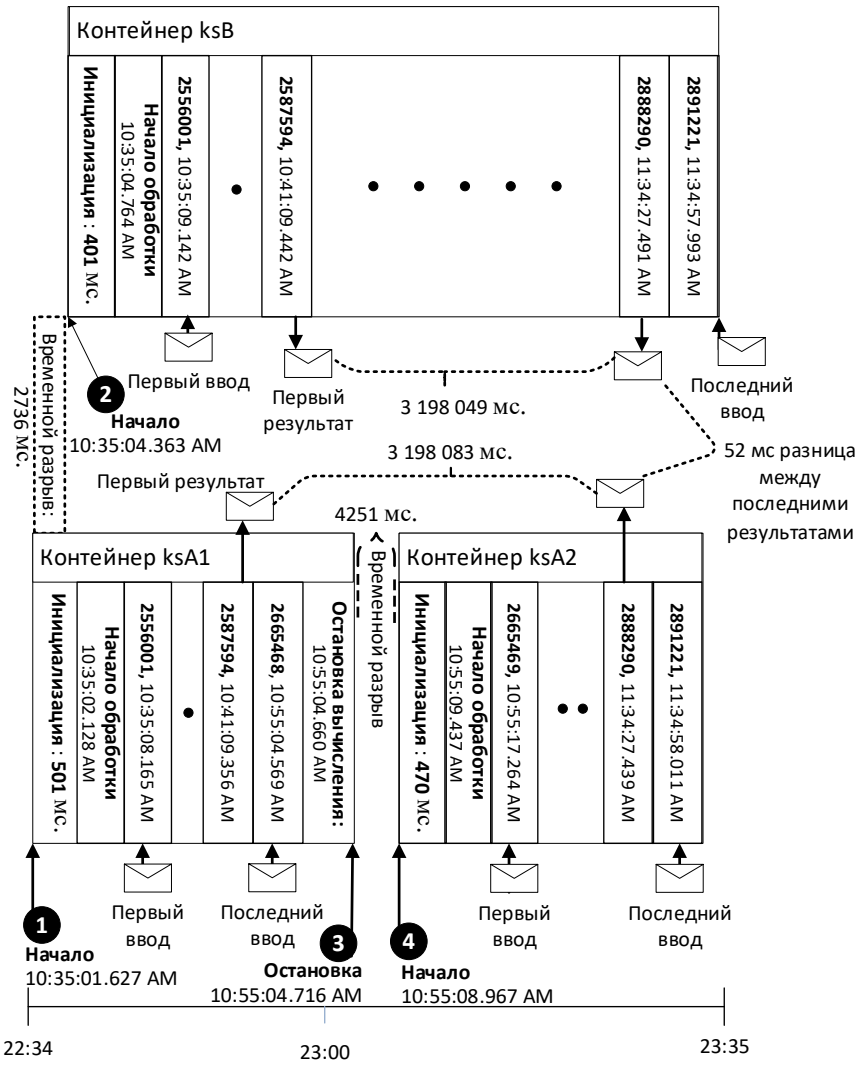


Рис. 38. Временная диаграмма проведения эксперимента по живой миграции.

идентификатором приложения, что и ksA1 (см. точку 4 на рис. 38). Таким образом, обеспечивается возможность ksA2 повторно использовать предыдущие промежуточные темы ksA1 для передачи состояния ksA1 в ksA2. Через 1 час генерация данных прекращается, и производится оценка результатов проведенного эксперимента.

На рис. 39, рис. 40, и рис. 41 приведены примеры загрузки процессора, входного и выходного трафика сети во время эксперимента. Они позволяют сравнить ход выполнения рабочей нагрузки как в непрерывном режиме (ksB), так и в режиме прерывания выполнения (ksA1, ksA2). Результаты прерывания вычислений можно наблюдать непосредственно перед отметкой

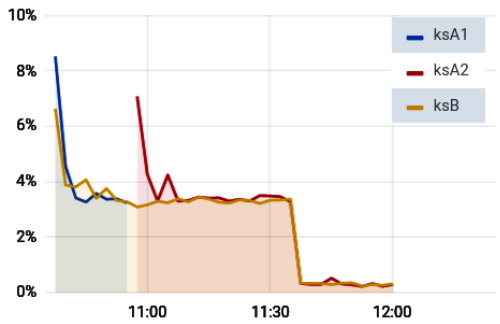


Рис. 39. График загрузки CPU

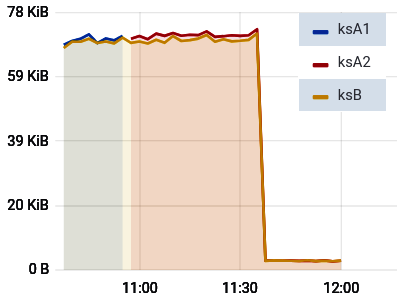


Рис. 40. График скорости входящего сетевого трафика

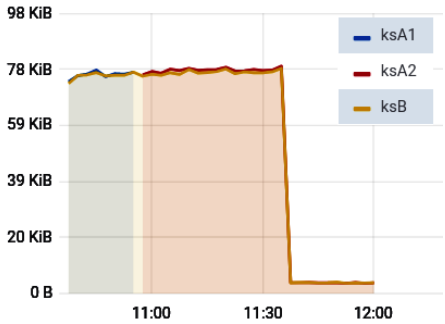


Рис. 41. График скорости исходящего сетевого трафика

11:00. График использования процессора показывает, что загрузка процессора при запуске прерванного задания выравнивается с той же скоростью, что и при запуске нового отдельного задания. Графики использования сети также не показывают существенных изменений в сетевой активности после прерывания работы: обработка данных полностью соответствует непрерывной работе.

4.5. Эксперимент по распределению вычислительной нагрузки в модели туманной вычислительной среды

Эксперимент исследует возможности развертывания вычислительной рабочей нагрузки на базе вычислительной среды, эмулирующей модель туманной вычислительной среды. В рамках эксперимента моделируется развертывание части вычислительных задач на географически-удаленных вычислительных узлах (например, на базе публичного облака), в то время как другая

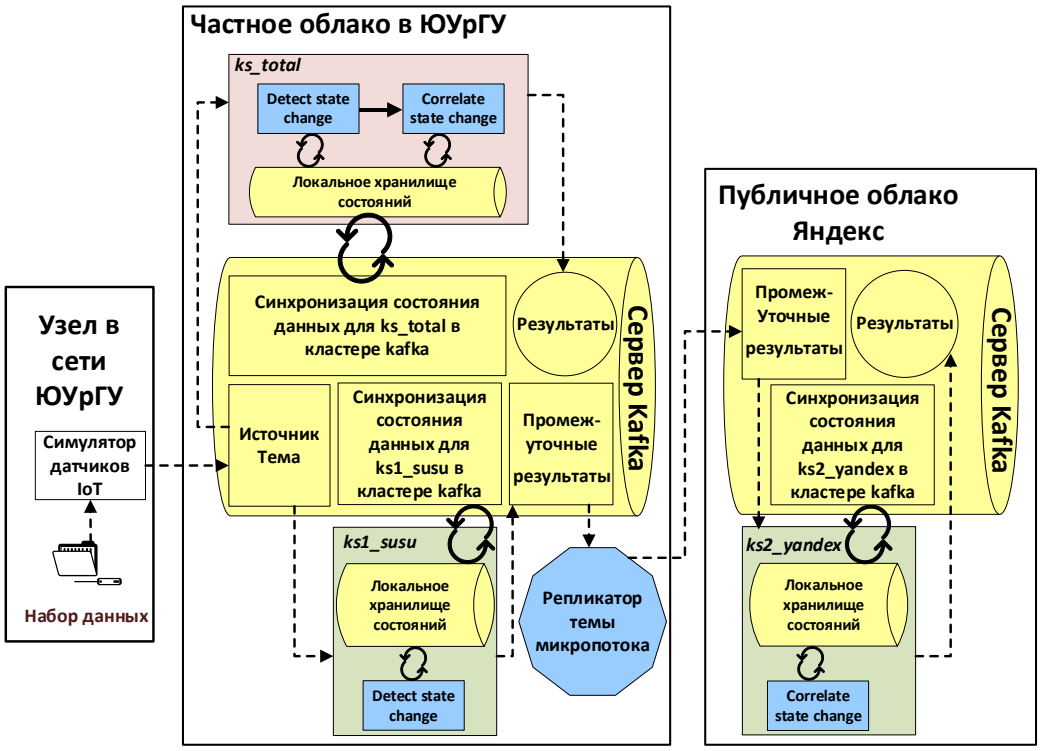


Рис. 42. Схема организации эксперимента по географическому распределению вычислительной нагрузки.

часть вычислительной рабочей нагрузки реализуется на узлах, находящихся рядом с источником данных, чтобы обеспечить минимальное время отклика.

В этом эксперименте сопоставляется два варианта обработки потока данных. *Испытание 1* представляет собой обработку данных в виде единого микро-потока работ (**ks_total**), развернутого в частном облаке ЮУрГУ. *Испытание 2* представляет собой обработку этого же потока данных посредством двух микро-потоков работ, один из которых (**ks1_susu**) размещен частном облаке ЮУрГУ, а второй (**ks2_yandex**) размещен в географически-удаленном вычислительном центре – публичном облаке компании Яндекс (см. рис. 42). Реализация потоков работ соответствует потокам работ, представленных в разделе 3.2. Обмен данными между центрами обработки данных организован через два географически разделенных кластера Kafka. Чтобы преодолеть проблемы репликации данных, которые все еще существуют в Kafka, был реализован репликатор для организации синхронизации географически разделенных кластеров Kafka (см. 3.4.2). На рис. 42 представлена

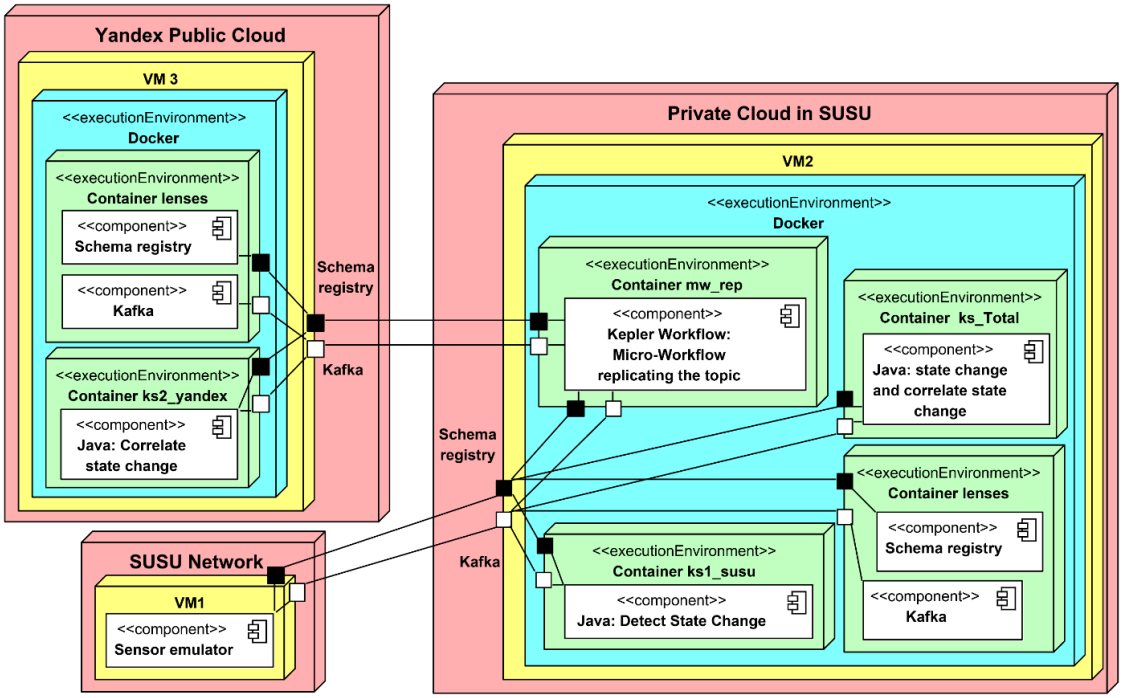


Рис. 43. Размещение компонентов эксперимента по географическому распределению вычислительной нагрузки.

схема организации эксперимента. Компоненты «Detect state change» и «Correlate state change» в ks_total, ks1_susu и ks2_yandex представляют собой вычислительные единицы, представляющие слой обработки данных (см. 1.3.3). Компоненты «Сервер Kafka» и «Хранилище локальных состояний» представляют собой слой хранения. В рамках данного эксперимента слой хранения также распределен: локальные хранилища состояний расположены непосредственно в контейнере, осуществляющем обработку данных.

В ходе эксперимента производится сравнение времени отклика, объема данных, передаваемых через глобальную вычислительную сеть, а также результаты выполнения обработки данных в рамках системы, состоящей из ks1_susu и ks2_yandex с результатами соответствующих частей обработки данных в ks_total.

Рис. 43 показывает детали развертывания компонентов эксперимента. Развертывание компонентов системы производится на трех узлах. Узел VM1 (4 ГБ оперативной памяти и 2-ядерный процессор Intel Xeon X5680)

Табл. 6. Сравнение результатов проведенных испытаний.

	Испытание 1	Испытание 2
Среднее время ответа на события «Detect state change»	75.71 мс.	72.38 мс.
Среднее время ответа на события «Correlate state change»	22 919 мс.	22 792 мс.

развернут в сети ЮУрГУ. На нем расположен источник данных и симулятор сенсоров. Узел VM2 (8 ГБ оперативной памяти и 8 ядер процессора Intel(R) Xeon(R) Gold 6242) развернут в частном облаке в ЮУрГУ. На нем размещен сервер Apache Kafka, контейнер ks_total, контейнер ks1_susu и контейнер mw_rep, который включает в репликатор данных обеспечивающий роутинг сообщений из темы в сервере Kafka на VM2 на сервер Kafka на VM3. Узел VM3 (6 ГБ оперативной памяти и 2-ядерный процессор Intel Cascade Lake) развернут в публичном облаке Яндекса. На нем размещен второй кластер Kafka и контейнер ks2_yandex. Задержка между VM1 и VM2 составляет в среднем 2 мс. Задержка между VM2 и VM3 составляет в среднем 30 мс.

Эксперимент состоит в организации обработки исходного потока данных в течение 6 часов. Как в рамках испытания 1, так и в рамках испытания 2, за это время было обработано 1 638 104 сообщения потока работ, что привело к генерации 1 616 результирующих сообщений. Средний промежуток времени между двумя последовательными сообщениями от симулятора датчиков составил 14 мс. Сравнение значений результатов обработки данных, полученных ks1_susu и ks2_yandex с результатами соответствующих частей в ks_total показало 100% соответствие результатов обработки данных с точки зрения полученных значений.

В табл. 6 приведены результаты проведенных испытаний. Среднее время ответа на событие «Detect state change» рассчитывается как усредненное время получения результатов в ks1_susu и соответствующей части в

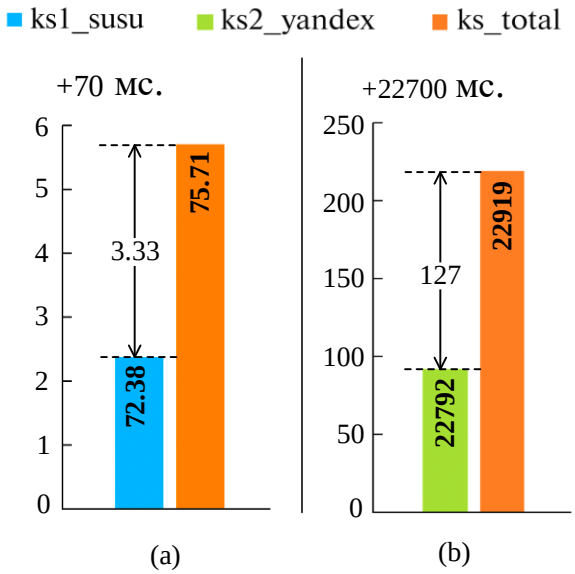


Рис. 44. Сравнение среднего времени ответа на событие: (a) «Detect state change» в ks1_susu и его соответствующей части в ks_total, (b) «Correlate state change» в ks2_yandex и его соответствующей части в ks_total.

ks_total по той же методологии, как значение метрики Av_TAT в рамках эксперимента, представленного в разделе 4.4.1. Среднее время ответа на событие «Correlate state change» рассчитывается как усредненное время получения результатов в ks2_susu и соответствующей части в ks_total. Существенная разница во времени получения результатов объясняется разной вычислительной сложностью, и объемом накладных расходов при обработке данных, вызванным реализацией механизма обработки данных с сохранением состояния. В первом случае («Detect state change»), требуется сравнение двух последовательных значений в потоке данных. В случае «Correlate state change», при получении каждого события реализуется перебор среди набора данных, переданных в потоке данных ранее.

На рис. 44 показано сравнение среднего времени получения ответа события «Detect state change» и «Correlate state change» соответственно. Результаты испытания 1 указаны как «ks_total». Результаты испытания 2 отмечены как «ks1_susu» и «ks2_yandex» соответственно. Улучшение временных параметров при реализации испытания 2 может быть объяснено следующим. В рамках испытания 1, и операции передачи промежуточных данных и

операции синхронизации состояния с сервером Apache Kafka, должны обрабатываться последовательно, для поддержки семантики обработки точно один раз (*exactly-once processing semantics*) (см. раздел 1.3.7). Это гарантирует, что каждая запись будет обработана один и только один раз, даже если в процессе обработки произойдет сбой на клиентах потоков или брокерах Apache Kafka. С другой стороны, разделение и распределение микро-потоков работ на независимые вычислительные микросервисы, как это реализовано в испытании 2, позволяет независимо управлять состоянием для каждого микро-потока. Apache Kafka рассматривает каждый микросервис как отдельное приложение, каждому из которых выделяется отдельные промежуточные темы потребления и генерации, что приводит к уменьшению накладных расходов при выполнении данных операций.

Также стоит отметить тот факт, что в результате предварительной обработки, реализуемой в рамках «ks1_susu», за время испытания 2 было сформировано 1 616 сообщений, что составляет менее 0.1% от начального объема потока данных, составляющего 1 638 104 сообщений. Учитывая, что размер сообщений, формируемых «ks1_susu» соответствует размеру входящих сообщений, размещение «ks1_susu» на узлах, расположенных в непосредственной близости к источнику данных позволило уменьшить объем данных, передаваемых в «ks2_yandex» по глобальной сети, в 1 014 раз по сравнению с вариантом, если бы вся обработка данных велась на узлах, размещенных в публичном облаке.

4.6. Вывод по главе 4

В главе 4 представлены результаты вычислительных экспериментов, по оценке применимости и эффективности концепции микро-потоков работ для обработки потоков данных в туманных вычислительных средах. В рамках эксперимента по рефакторингу потока работ было проведено испытание и проверка программной утилиты, реализующей алгоритм рефакторинга

(см. 2.3), обеспечивающего разделение монолитного потока работ на набор независимых микро-потоков работ. В рамках эксперимента по сравнению по сравнению монолитного потока работ и микро-потока работ для обработки потоков данных в реальном времени было показано преимущество концепции микро-потоков работ в плане уменьшения времени отклика на события, которые могут быть идентифицированы в потоке данных IoT. Эксперименты по локальному и распределенному развертыванию, а также по возможностям развертывания микро-потоков работ в модели туманной вычислительной среды показали возможность применения концепции микро-потоков работ для распределения вычислительной нагрузки между слоями географически-распределенных вычислительных систем, таких как туманные вычислительные системы, с целью минимизации времени отклика. Эксперименты по обработке данных с сохранением состояния показали возможности использования микро-потоков работ в условиях преднамеренной либо непреднамеренной остановки вычислительных сервисов. Это позволяет говорить о применимости данной концепции в нестабильных и динамически-изменяющихся условиях туманных и облачных вычислительных сред.

Результаты экспериментов показывают, что концепция микро-потоков работ расширяет возможности существующих подходов благодаря сочетанию мощности научных потоков работ, гибкости контейнерных технологий и устойчивости подхода потоковой передачи данных. Она поддерживает возможность обработки событий от различных источников (таких как датчики IoT) внутри вычислительных потоков работ. Внедрение и тестирование данной архитектуры на базе системы управления научными потоками работ Kepler и платформы потоковой обработки данных Apache Kafka показывают, что предлагаемая архитектура успешно решает задачи обработки данных интернета вещей, обеспечивая достаточно небольшие размеры сетевой задержки. Все эксперименты и результаты, включенные в эту главу, опубликованы в работах [5–10,85,123,124].

ЗАКЛЮЧЕНИЕ

В диссертационной работе были рассмотрены вопросы разработки и исследования моделей обработки потоков данных, обеспечивающих поддержку для реализации приложений в туманных вычислительных средах. Основным научным результатом является создание концепции микро-потоков работ, позволяющей организовать эффективную обработку потоков данных в туманных вычислительных средах с применением потоков работ. Разработан алгоритм рефакторинга монолитных приложений потоков работ в наборы микро-потоков работ. Разработан на языке Java комплекс вычислительных акторов и программных утилит для поддержки функционирования микро-потоков работ на базе платформы управления потоками работ Kepler и платформы обработки потоков данных Apache Kafka. С использованием реализованных акторов и программных утилит были проведены вычислительные эксперименты для верификации предложенной модели и разработанного программного обеспечения.

Основные результаты, полученные в ходе выполнения диссертационного исследования, являются новыми и не покрываются ранее опубликованными научными работами других авторов, обзор которых был дан в разделе 1.5. Отметим основные отличия:

1) областью применения подели микро-потоков работ является интеграция поддержки поточной обработки данных в приложения потоков работ; ни одна из известных моделей декомпозиции потоков работ на под-потоки работ не ориентирована на реализацию такого формата обработки данных;
2) модель микро-потоков работ позволяет на порядки уменьшить время задержки получения результата при обработке потоков данных по сравнению с пакетными подходами обработки данных;
3) ни одна из рассмотренных работ не включала в себя алгоритм декомпозиции потока работ на набор независимых микро-потоков работ,

взаимодействие между которыми осуществляется на основе событийно-ориентированной модели, описанный в разделе 2.3;

4) в отличие от типовых решений обработки потоков данных, таких как Spark, Storm и Kafka, применение концепции микро-потоков работ позволяет интегрировать в туманные вычислительные среды существующие вычислительные процессы, реализованные на основе потоков работ;
5) модель микро-потоков работ не применима в случае организации обработки больших объемов данных в пакетном режиме в связи с накладными расходами, связанными с необходимостью постоянного выполнения вычислительных сервисов микро-потоков работ на узлах вычислительной системы, а также поддержкой обмена данными с сохранением состояния.

Разработанная в ходе настоящего диссертационного исследования модель микро-потоков работ и алгоритм рефакторинга потоков работ на независимые микро-потоки работ могут применяться для организации вычислительного процесса обработки потоков данных в туманных вычислительных средах. Разработанный комплекс вычислительных акторов и программных утилит для поддержки функционирования микро-потоков работ может быть использован для создания микро-потоков работ на базе системы Kepler и платформы обработки потоков данных Apache Kafka.

В качестве направления дальнейших исследований можно выделить разработку методов автоматизированного планирования и масштабирования микро-потоков работ в туманных вычислительных средах.

Работа выполнена при финансовой поддержке РФФИ в рамках научных проектов № 19-37-90073 и № 18-07-01224, а также при поддержке Министерства науки и высшего образования РФ в рамках государственного задания № FENU-2020-0022.

ЛИТЕРАТУРА

1. Aazam M., Zeadally S., Harras K.A. Deploying Fog Computing in Industrial Internet of Things and Industry 4.0 // IEEE Transactions on Industrial Informatics. IEEE, 2018. Vol. 14, no. 10. P. 4674–4682. DOI:10.1109/TII.2018.2855198.
2. Abdollahi Vayghan L. et al. Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes // Proceedings - 19th IEEE International Conference on Software Quality, Reliability and Security, QRS 2019. IEEE, 2019. P. 176–185. DOI:10.1109/QRS.2019.00034.
3. Agneeswaran V.S. Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternatives. 2014.
4. Ahn H., Kim K.P. Formal Approach to Workflow Application Fragmentations over Cloud Deployment Models // Computers, Materials and Continua. 2021. Vol. 67, no. 3. P. 3071–3088. DOI:10.32604/cmc.2021.015280.
5. Alaasam A.B.A., Radchenko G.I., Tchernykh A.N. Stateful Stream Processing for IoT Systems // Двенадцатая научная конференция аспирантов и докторантов “научный поиск” 17-19 марта 2020 - Секция: естественные науки - Южно-Уральский государственный университет (ЮурГУ). Челябинск, Россия: Южно-Уральский государственный университет (ЮурГУ) - Издательский центр ЮУрГУ (Челябинск), 2020. P. 36–41.
6. Alaasam A.B.A. et al. Scientific Micro-Workflows : Where Event-Driven Approach Meets Workflows to Support Digital Twins // Proceedings of the international conference RuSCDays’18 - Russian Supercomputing Days (September 24-25, 2018, Moscow, Russia), MSU. 2018. Vol. 1. P. 489–495.
7. Alaasam A.B.A. et al. Analytic Study of Containerizing Stateful Stream Processing as Microservice to Support Digital Twins in Fog Computing // Programming and Computer Software. 2020. Vol. 46, no. 8. P. 511–525. DOI:10.1134/S0361768820080083.
8. Alaasam A.B.A., Radchenko G.I., Tchernykh A.N. Micro-Workflows Data Stream Processing Model for Industrial Internet of Things // Supercomputing Frontiers and Innovations. 2021. Vol. 8, no. 1. P. 82–98. DOI:10.14529/jsfi210106.
9. Alaasam A.B.A., Radchenko G., Tchernykh A. Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL // 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). IEEE, 2019. P. 0804–0809. DOI:10.1109/SIBIRCON48586.2019.8958367.

10. Alaasam A.B.A., Radchenko G., Tchernykh A. Refactoring the Monolith Workflow into Independent Micro-Workflows to Support Stream Processing // Programming and Computer Software. 2021. Vol. 47, no. 8. P. 591–600. DOI:10.1134/S0361768821080077.
11. Altintas I. et al. Kepler: an extensible system for design and execution of scientific workflows // Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004. 2004. P. 423–424. DOI:10.1109/SSDM.2004.1311241.
12. Anderson D.P. BOINC: A System for Public-Resource Computing and Storage // Fifth IEEE/ACM International Workshop on Grid Computing. IEEE, 2004. P. 4–10. DOI:10.1109/GRID.2004.14.
13. Andrade H., Gedik B., Turaga D. Fundamentals of Stream Processing // Fundamentals of Stream Processing. Cambridge: Cambridge University Press, 2014. DOI:10.1017/CBO9781139058940.
14. Antonić A. et al. A high throughput processing engine for taxi-generated data streams // Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems - DEBS ’15. New York, New York, USA: ACM Press, 2015. P. 309–315. DOI:10.1145/2675743.2772588.
15. Badia R.M., Ayguade E., Labarta J. Workflows for science: a challenge when facing the convergence of HPC and Big Data // Supercomputing Frontiers and Innovations. 2017. Vol. 4, no. 1. P. 27–47. DOI:10.14529/jsfi170102.
16. Balta E.C., Tilbury D.M., Barton K. A Digital Twin Framework for Performance Monitoring and Anomaly Detection in Fused Deposition Modeling // 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE). IEEE, 2019. no. 70. P. 823–829. DOI:10.1109/COASE.2019.8843166.
17. Barga R., Gannon D. Scientific versus Business Workflows // Workflows for e-Science. London: Springer London, 2007. P. 9–16. DOI:10.1007/978-1-84628-757-2_2.
18. Bellavista P., Zanni A. Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi // Proceedings of the 18th International Conference on Distributed Computing and Networking. New York, NY, USA: ACM, 2017. P. 1–10. DOI:10.1145/3007748.3007777.
19. Bonomi F. et al. Fog computing and its role in the internet of things // MCC’12 - Proceedings of the 1st ACM Mobile Cloud Computing Workshop. 2012. P. 13–15. DOI:10.1145/2342509.2342513.
20. Borodulin K. et al. Towards Digital Twins Cloud Platform : Microservices and Computational Workflows to Rule a Smart Factory // Proceedings of the10th International Conference on Utility and Cloud Computing - UCC ’17. New York, New York, USA: ACM Press, 2017. no. December. P. 209–210. DOI:10.1145/3147213.3149234.

21. Carvalho O., Roloff E., Navaux P.O.A. A Distributed Stream Processing based Architecture for IoT Smart Grids Monitoring // Companion Proceedings of the10th International Conference on Utility and Cloud Computing. New York, NY, USA: ACM, 2017. P. 9–14. DOI:10.1145/3147234.3148105.
22. Chandy K.M. Event Driven Architecture // Encyclopedia of Database Systems. Boston, MA: Springer US, 2009. P. 1040–1044. DOI:10.1007/978-0-387-39940-9_570.
23. Clark C. et al. Live Migration of Virtual Machines // NSDI'05 Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2. 2005. P. 273–286.
24. Deelman E. et al. Pegasus, a workflow management system for science automation // Future Generation Computer Systems. Elsevier B.V., 2015. Vol. 46. P. 17–35. DOI:10.1016/j.future.2014.10.008.
25. Dias de Assunção M., da Silva Veith A., Buyya R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions // Journal of Network and Computer Applications. Elsevier Ltd, 2018. Vol. 103. P. 1–17. DOI:10.1016/j.jnca.2017.12.001.
26. Ericsson. Ericsson Mobility Report: On The Pulse Of The Networked Society // White Paper. 2015. no. November. P. 1–36. DOI:10.3103/S0005105510050031.
27. Fahringer T. et al. ASKALON: a Grid application development and computing environment // The 6th IEEE/ACM International Workshop on Grid Computing, 2005. 2005. P. 10 pp. DOI:10.1109/GRID.2005.1542733.
28. Feeney G.J. et al. Utility computing: a superior alternative? // Proceedings of the May 6-10, 1974, national computer conference and exposition on - AFIPS '74. New York, New York, USA: ACM Press, 1974. P. 1003. DOI:10.1145/1500175.1500370.
29. Fiore M., Devesas Campos M. The Algebra of Directed Acyclic Graphs // Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky. Springer Berlin Heidelberg, 2013. P. 37–51. DOI:10.1007/978-3-642-38164-5_4.
30. Foster I., Kesselman C. Globus: a Metacomputing Infrastructure Toolkit // The International Journal of Supercomputer Applications and High Performance Computing. 1997. Vol. 11, no. 2. P. 115–128. DOI:10.1177/109434209701100205.
31. Garfinkel S.L. Architects of the Information Society, Thirty-Five Years of the Laboratory for Computer Science at MIT / ed. Harold Abelson. Cambridge: MIT Press, 1999.
32. Gavaldà R. Adaptive Windowing // Encyclopedia of Big Data Technologies. Cham: Springer International Publishing, 2018. P. 1–6. DOI:10.1007/978-3-319-63962-8_194-1.

33. Gholami M.F. et al. Cloud migration process—A survey, evaluation framework, and open challenges // Journal of Systems and Software. Elsevier Inc., 2016. Vol. 120. P. 31–69. DOI:10.1016/j.jss.2016.06.068.
34. Glaessgen E.H., Stargel D.D.S. The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles // 53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference - Special Session on the Digital Twin. Honolulu, HI, United States: American Institute of Aeronautics and Astronautics, 2012. no. April. P. 1–14. DOI:10.2514/6.2012-1818.
35. Golab L. Types of Stream Processing Algorithms // Encyclopedia of Big Data Technologies. Cham: Springer International Publishing, 2019. P. 1726–1732. DOI:10.1007/978-3-319-77525-8_193.
36. Golab L., Özsu M.T. Issues in data stream management // SIGMOD Record. 2003. Vol. 32, no. 2. P. 5–14. DOI:10.1145/776985.776986.
37. Goyal P., Mikkilineni R. Policy-Based Event-Driven Services-Oriented Architecture for Cloud Services Operation & Management // 2009 IEEE International Conference on Cloud Computing. IEEE, 2009. P. 135–138. DOI:10.1109/CLOUD.2009.76.
38. Grieves M., Vickers J. Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems // Transdisciplinary Perspectives on Complex Systems. Cham: Springer International Publishing, 2017. no. August 2017. P. 85–113. DOI:10.1007/978-3-319-38756-7_4.
39. Gualtieri M., Yuhanna N. The forrester wave: Big data streaming analytics, Q1 2016 // Forrester research. Cambridge, MA, USA, 2016. 15 p.
40. Gubbi J. et al. Internet of Things (IoT): A vision, architectural elements, and future directions // Future Generation Computer Systems. 2013. Vol. 29, no. 7. P. 1645–1660. DOI:10.1016/j.future.2013.01.010.
41. Haag S., Anderl R. Digital twin – Proof of concept // Manufacturing Letters. Society of Manufacturing Engineers (SME), 2018. Vol. 15, no. June. P. 64–66. DOI:10.1016/j.mfglet.2018.02.006.
42. Harju T. Lecture Notes on Graph Theory. University of Turku, Finland, 2014.
43. Hassan R. et al. Internet of things and its applications: A comprehensive survey // Symmetry. 2020. Vol. 12, no. 10. P. 1–29. DOI:10.3390/sym12101674.
44. Henning S., Hasselbring W. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures // Big Data Research. 2021. Vol. 25. DOI:10.1016/j.bdr.2021.100209.
45. Hirales-Carbajal A. et al. A Grid simulation framework to study advance scheduling strategies for complex workflow applications // Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010. IEEE, 2010. P. 1–8. DOI:10.1109/IPDPSW.2010.5470918.

46. Hirales-Carbajal A. et al. Multiple workflow scheduling strategies with user run time estimates on a Grid // Journal of Grid Computing. 2012. Vol. 10, no. 2. P. 325–346. DOI:10.1007/s10723-012-9215-6.
47. Hirales-Carbajal A. et al. Multiple Workflow Scheduling Strategies with User Run Time Estimates on a Grid // Journal of Grid Computing. 2012. Vol. 10, no. 2. P. 325–346. DOI:10.1007/s10723-012-9215-6.
48. Hoque S. et al. Towards Container Orchestration in Fog Computing Infrastructures // Proceedings - International Computer Software and Applications Conference. 2017. Vol. 2. P. 294–299. DOI:10.1109/COMPSAC.2017.248.
49. Iorga M. et al. Fog computing conceptual model. Gaithersburg, MD, 2018. DOI:10.6028/NIST.SP.500-325.
50. Isah H. et al. A Survey of Distributed Data Stream Processing Frameworks // IEEE Access. IEEE, 2019. Vol. 7, no. October. P. 154300–154316. DOI:10.1109/ACCESS.2019.2946884.
51. Iturriaga S. et al. Multiobjective Workflow Scheduling in a Federation of Heterogeneous Green-Powered Data Centers // 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). IEEE, 2016. no. October 2017. P. 596–599. DOI:10.1109/CCGrid.2016.34.
52. James Lewis, Martin Fowler. Microservices [Electronic resource]. 2014. URL: https://martinfowler.com/articles/microservices.html (accessed: 11.01.2019).
53. Jing Han et al. Survey on NoSQL database // 2011 6th International Conference on Pervasive Computing and Applications. IEEE, 2011. P. 363–366. DOI:10.1109/ICPCA.2011.6106531.
54. Kambatla K. et al. Trends in big data analytics // Journal of Parallel and Distributed Computing. 2014. Vol. 74, no. 7. P. 2561–2573. DOI:10.1016/j.jpdc.2014.01.003.
55. Kevin Asthon. That ’ Internet of Things ’ Thing // RFID Journal. 1999. P. 4986.
56. v. Knyazkov K. et al. CLAVIRE: e-Science infrastructure for data-driven computing // Journal of Computational Science. 2012. Vol. 3, no. 6. P. 504–510. DOI:10.1016/j.jocs.2012.08.006.
57. Korambath P. et al. Deploying kepler workflows as services on a cloud infrastructure for smart manufacturing // Procedia Computer Science. Elsevier Masson SAS, 2014. Vol. 29. P. 2254–2259. DOI:10.1016/j.procs.2014.05.210.
58. Korambath P. et al. A Smart Manufacturing Use Case: Furnace Temperature Balancing in Steam Methane Reforming Process via Kepler Workflows // Procedia Computer Science. Elsevier Masson SAS, 2016. Vol. 80. P. 680–689. DOI:10.1016/j.procs.2016.05.357.
59. Liew C.S. et al. Scientific Workflows: Moving Across Paradigms // ACM Computing Surveys. 2017. Vol. 49, no. 4. P. 1–39. DOI:10.1145/3012429.

60. Lin B. et al. A Pretreatment Workflow Scheduling Approach for Big Data Applications in Multicloud Environments // IEEE Transactions on Network and Service Management. Institute of Electrical and Electronics Engineers Inc., 2016. Vol. 13, no. 3. P. 581–594. DOI:10.1109/TNSM.2016.2554143.
61. Litzkow M.J., Livny M., Mutka M.W. Condor - a hunter of idle workstations // [1988] Proceedings. The 8th International Conference on Distributed. IEEE Comput. Soc. Press, 1988. Vol. 8, no. December. P. 104–111. DOI:10.1109/DCS.1988.12507.
62. Liu J. et al. A Survey of Data-Intensive Scientific Workflow Management // Journal of Grid Computing. 2015. Vol. 13, no. 4. P. 457–493. DOI:10.1007/s10723-015-9329-8.
63. Ludäscher B. et al. Scientific workflow management and the Kepler system // Concurrency and Computation: Practice and Experience. John Wiley & Sons, Ltd., 2006. Vol. 18, no. 10. P. 1039–1065. DOI:10.1002/cpe.994.
64. Luo J. et al. Container-based fog computing architecture and energy-balancing scheduling algorithm for energy IoT // Future Generation Computer Systems. Elsevier B.V., 2019. Vol. 97. P. 50–60. DOI:10.1016/j.future.2018.12.063.
65. Madni A., Madni C., Lucero S. Leveraging Digital Twin Technology in Model-Based Systems Engineering // Systems. 2019. Vol. 7, no. 1. P. 7. DOI:10.3390/systems7010007.
66. Marcu O.-C. et al. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks // 2016 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2016. P. 433–442. DOI:10.1109/CLUSTER.2016.22.
67. Margara A., Rabl T. Definition of Data Streams // Encyclopedia of Big Data Technologies. Cham: Springer International Publishing, 2019. P. 648–652. DOI:10.1007/978-3-319-77525-8_188.
68. Mark Richards. Software Architecture Patterns. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2015.
69. Meehan J., Zdonik S. Data Ingestion for the Connected World // Cidr. 2017.
70. Meng X. et al. A data-intensive workflow scheduling algorithm for grid computing // 4th ChinaGrid Annual Conference, ChinaGrid 2009. 2009. P. 110–115. DOI:10.1109/ChinaGrid.2009.30.
71. Modoni G.E., Sacco M., Terkaj W. A Telemetry-driven Approach to Simulate Data-intensive Manufacturing Processes // Procedia CIRP. 2016. Vol. 57. P. 281–285. DOI:10.1016/j.procir.2016.11.049.
72. Morales J.A.S.-C., Torres-ramos S. Dynamic Communication-Aware Scheduling with Uncertainty of Workflow Applications in Clouds // High Performance Computer Applications / ed. Gitler I., Klapp J. Cham: Springer International Publishing, 2016. Vol. 595, no. October 2017. P. 169–187. DOI:10.1007/978-3-319-32243-8_12.

73. Mutka M.W., Livny M. Profiling Workstations Available Capacity for Remote Execution // Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation. 1987. no. May. P. 529--544.
74. Naseri M., Towhidi A. Stateful Web Services: A Missing Point in Web Service Standards // Proceedings of the International MultiConference of Engineers and Computer Scientists 2007 (IMECS 2007). Hong Kong, China, 2007. P. 993–997.
75. Newman S. Building Microservices: Designing Fine-Grained System. O'Reilly Media, 2015. 280 p.
76. Niqui M., Rutten J. Sampling, Splitting and Merging in Coinductive Stream Calculus. 2010. P. 310–330. DOI:10.1007/978-3-642-13321-3_18.
77. Ohtsuji H., Tatebe O. Network-Based Data Processing Architecture for Reliable and High-Performance Distributed Storage System. 2015. P. 16–26. DOI:10.1007/978-3-319-27308-2_2.
78. Ozeer U. et al. Resilience of Stateful IoT Applications in a Dynamic Fog Environment // Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. New York, NY, USA: ACM, 2018. P. 332–341. DOI:10.1145/3286978.3287007.
79. Parrott A., Lane W. Industry 4.0 and the digital twin: Manufacturing meets its match // Deloitte University Press. 2017. P. 1–17.
80. Peiffer C., L'Heureux I. System and method for maintaining statefulness during client-server interactions: US8346848B2 // (12) United States Patent. United States of America, 2013. no. US8346848B2.
81. Perry Lea. Internet of Things for Architects. Packt Publishing Ltd., 2018. 524 p.
82. Plociennik M. et al. Approaches to Distributed Execution of Scientific Workflows in Kepler // Fundamenta Informaticae. 2013. Vol. 128, no. 3. P. 281–302. DOI:10.3233/FI-2013-947.
83. Pollock B.E. et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems // Scientific Programming. 1997. Vol. 86, no. 2. P. 319–320; author reply 320-1.
84. Qin J., Fahringer T. Scientific Workflows // Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Vol. 9783642307. 1–222 p. DOI:10.1007/978-3-642-30715-7.
85. Radchenko G., Alaasam A.B.A., Tchernykh A. Micro-Workflows: Kafka and Kepler Fusion to Support Digital Twins of Industrial Processes // 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). Zurich, Switzerland: IEEE, 2018. no. 18. P. 83–88. DOI:10.1109/UCC-Companion.2018.00039.

86. Radchenko G., Hudyakova E. A service-oriented approach of integration of computer-aided engineering systems in distributed computing environments // UNICORE Summit 2012, Proceedings. 2012. Vol. 15. P. 57–66.
87. Radchenko G.I., Alaasam A.B.A., Tchernykh A.N. Comparative Analysis of Virtualization Methods in Big Data Processing // Supercomputing Frontiers and Innovations. 2019. Vol. 6, no. 1. P. 48–79. DOI:10.14529/jsfi190107.
88. Reber A. CRIU and the PID dance // Linux Plumbers Conference 2019. 2019. P. 1–4.
89. Romero C., Oliveira H.P. Kafka: a Distributed Messaging System for Log Processing // Proceedings of 6th international workshop on networking meets databases (NetDB). Athens, Greece, 2011.
90. Sakellariou R., Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems // 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. IEEE, 2004. Vol. 18. P. 111–123. DOI:10.1109/IPDPS.2004.1303065.
91. Savchenko D.I., Radchenko G.I., Taipale O. Microservices validation: Mjolnirr platform case study // 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, 2015. P. 235–240. DOI:10.1109/MIPRO.2015.7160271.
92. Scheibmeir J., Malaiya Y. An API Development Model for Digital Twins // 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2019. P. 518–519. DOI:10.1109/QRS-C.2019.00103.
93. Shabanov B.M., Samovarov O.I. Building the Software-Defined Data Center // Programming and Computer Software. 2019. Vol. 45, no. 8. P. 458–466. DOI:10.1134/S0361768819080048.
94. Shahrivari S. Beyond batch processing: Towards real-time and streaming big data // Computers. 2014. Vol. 3, no. 4. P. 117–129. DOI:10.3390/computers3040117.
95. Silva R.F. da et al. WorkflowHub: Community Framework for Enabling Scientific Workflow Research and Development // 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). IEEE, 2020. P. 49–56. DOI:10.1109/WORKS51914.2020.00012.
96. Singh S. et al. Streaming Machine Generated Data to Enable a Third-Party Ecosystem of Digital Manufacturing Apps // Procedia Manufacturing. The Author(s), 2017. Vol. 10, no. Dmc. P. 1020–1030. DOI:10.1016/j.promfg.2017.07.093.
97. Smarr L., Catlett C.E. Metacomputing // Communications of the ACM. 1992. Vol. 35, no. 6. P. 44–52. DOI:10.1145/129888.129890.

98. Smirnov P., Melnik M., Nasonov D. Performance-aware scheduling of streaming applications using genetic algorithm // Procedia Computer Science. 2017. Vol. 108. P. 2240–2249. DOI:10.1016/j.procs.2017.05.249.
99. Streit A. et al. Unicore — From project results to production grids // Advances in Parallel Computing. 2005. Vol. 14, no. C. P. 357–376. DOI:10.1016/S0927-5452(05)80018-8.
100. Sukhoroslov O. Toward efficient execution of data-intensive workflows // The Journal of Supercomputing. 2021. Vol. 77, no. 8. P. 7989–8012. DOI:10.1007/s11227-020-03612-4.
101. Sunderrajan A., Aydt H., Knoll A. DEBS Grand Challenge : Real time Load Prediction and Outliers Detection using STORM // DEBS ’14 Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. Mumbai, India, 2014. P. 294–297.
102. Talia D., Trunfio P., Marozzo F. Designing and Supporting Scalable Data Analytics // Data Analysis in the Cloud. Elsevier, 2016. P. 77–122. DOI:10.1016/B978-0-12-802881-0.00004-4.
103. Tan W., Fan Y. Dynamic workflow model fragmentation for distributed execution // Computers in Industry. 2007. Vol. 58, no. 5. P. 381–391. DOI:10.1016/j.compind.2006.07.004.
104. Taneja M. et al. SmartHerd management: A microservices-based fog computing–assisted IoT platform towards data-driven smart dairy farming // Software: Practice and Experience. John Wiley and Sons Ltd, 2019. Vol. 49, no. 7. P. 1055–1078. DOI:10.1002/spe.2704.
105. Taneja M., Davy A. Resource aware placement of IoT application modules in Fog-Cloud Computing Paradigm // 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). IEEE, 2017. no. May. P. 1222–1228. DOI:10.23919/INM.2017.7987464.
106. Tchernykh A. et al. Scalable Data Storage Design for Non-Stationary IoT Environment with Adaptive Security and Reliability // IEEE Internet of Things Journal. 2020. no. 61363019. P. 1–1. DOI:10.1109/JIOT.2020.2981276.
107. Teslyuk A. et al. Development of Experimental Data Processing Workflows Based on Kubernetes Infrastructure and REANA Workflow Management System. 2020. P. 563–573. DOI:10.1007/978-3-030-64616-5_48.
108. Trilles S. et al. Real-Time Anomaly Detection from Environmental Data Streams // Lecture Notes in Geoinformation and Cartography. 2015. Vol. 217. P. 125–144. DOI:10.1007/978-3-319-16787-9_8.
109. US Census Bureau world population. Population Clock World [Electronic resource]. 2019. URL: https://www.census.gov/popclock/world (accessed: 30.11.2019).
110. Vaquero L.M. et al. A break in the clouds: towards a cloud definition // ACM SIGCOMM Computer Communication Review. 2008. Vol. 39, no. 1. P. 50–55. DOI:10.1145/1496091.1496100.

111. Verma A. et al. Large-scale cluster management at Google with Borg // Proceedings of the Tenth European Conference on Computer Systems. New York, NY, USA: ACM, 2015. P. 1–17. DOI:10.1145/2741948.2741964.
112. Voevodin Vl.V. The solution of large problems in distributed computational media // Automation and Remote Control. 2007. Vol. 68, no. 5. P. 773–786. DOI:10.1134/S0005117907050050.
113. Wang J., Crawl D., Altintas I. Kepler + Hadoop: A General Architecture Facilitating Data-Intensive Applications in Scientific Workflow Systems // Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science - WORKS '09. 2009. P. 1–8. DOI:10.1145/1645164.1645176.
114. Wang L., Wang G., Alexander C.A. Confluences among Big Data, Finite Element Analysis and High Performance Computing // American Journal of Engineering and Applied Sciences. 2015. Vol. 8, no. 4. P. 767–774. DOI:10.3844/ajeassp.2015.767.774.
115. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management // Journal of Circuits, Systems and Computers. 1998. Vol. 08, no. 01. P. 21–66. DOI:10.1142/S0218126698000043.
116. W.M.P. van der Aalst, ter Hofstede A.H.M. Verification of Workflow Task Structures: A Petri-net-baset Approach // Information Systems. 2000. Vol. 25, no. 1. P. 43–69. DOI:10.1016/S0306-4379(00)00008-9.
117. Woods D., Mattern T. Enterprise SOA: Designing IT for Business Innovation. O'Reilly Media, 2006.
118. Xu Q. et al. Building a large-scale object-based active storage platform for data analytics in the internet of things // Journal of Supercomputing. Springer US, 2016. Vol. 72, no. 7. P. 2796–2814. DOI:10.1007/s11227-016-1621-2.
119. Yang P.-C. et al. A demonstration of modularity, reuse, reproducibility, portability and scalability for modeling and simulation of cardiac electrophysiology using Kepler Workflows // PLOS Computational Biology / ed. Sauro H. 2019. Vol. 15, no. 3. P. e1006856. DOI:10.1371/journal.pcbi.1006856.
120. Zhang Q., Cheng L., Boutaba R. Cloud computing: state-of-the-art and research challenges // Journal of Internet Services and Applications. 2010. Vol. 1, no. 1. P. 7–18. DOI:10.1007/s13174-010-0007-6.
121. Zhao Y. et al. Scientific-workflow-management-as-a-service in the cloud // Proceedings - 2nd International Conference on Cloud and Green Computing and 2nd International Conference on Social Computing and Its Applications, CGC/SCA 2012. 2012. P. 97–104. DOI:10.1109/CGC.2012.70.
122. Zheng C., Tovar B., Thain D. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos // Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and

Grid Computing, CCGRID 2017. 2017. no. 2. P. 130–139. DOI:10.1109/CCGRID.2017.9.
123. 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 Института системного программирования РАН. 2021. Т. 33, № 1. С. 65–80. DOI:10.15514/ISPRAS-2021-33(1)-5.
124. Алаасам А.Б.А., Радченко Г.И., Черных А.Н. Микро-потоки работ: сочетание потоков работ и потоковой обработки данных для поддержки цифровых двойников технологических процессов // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 4. С. 100–116. DOI:10.14529/cmse190407.
125. Буркатовская Ю.Б. Теория графов. Издательство Томского политехнического университета, 2014.
126. Воеводин Вл.В., Жолудев Ю.А., Соболев С.И., Стефанов К.С. Эволюция системы метакомпьютинга X-Com // Вестник Нижегородского университета им. Н.И. Лобачевского. 2009. Т. 4. С. 157–164.
127. Sensor Market Size, Share, Trends and Industry Analysis by 2025 | AMR [Electronic resource]. URL: https://www.alliedmarketresearch.com/sensor-market (accessed: 30.11.2019).
128. Global Autonomous Vehicle Market- Industry Trends & Forecast Report 2027 [Electronic resource]. URL: https://www.blueweaveconsulting.com/global-autonomous-vehicles-market (accessed: 14.03.2021).
129. Driverless Car Data Storage | Automakers, Suppliers Grapple Overflow | WardsAuto [Electronic resource]. URL: https://www.wardsauto.com/technology/storage-almost-full-driverless-cars-create-data-crunch (accessed: 14.03.2021).
130. MapReduce Tutorial [Electronic resource]. URL: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html (accessed: 17.10.2021).
131. Apache Avro™ 1.10.2 Getting Started (Java) [Electronic resource]. URL: https://avro.apache.org/docs/current/gettingstartedjava.html (accessed: 24.04.2021).
132. Real-time full-text search with Luwak and Samza - Confluent [Electronic resource]. URL: https://www.confluent.io/blog/real-time-full-text-search-with-luwak-and-samza/ (accessed: 27.04.2021).
133. Live migration - CRIU [Electronic resource]. URL: https://criu.org/Live_migration (accessed: 23.12.2019).
134. Server Management Software - vCenter Server | VMware [Electronic resource]. URL: https://www.vmware.com/products/vcenter-server.html (accessed: 11.12.2019).
135. About storage drivers | Docker Documentation [Electronic resource]. URL: https://docs.docker.com/storage/storagedriver/ (accessed: 15.12.2019).

136. docker checkpoint | Docker Documentation [Electronic resource]. URL: https://docs.docker.com/engine/reference/commandline/checkpoint/ (accessed: 11.12.2019).
137. Docker - CRIU [Electronic resource]. URL: https://criu.org/Docker (accessed: 21.03.2021).
138. LXD 2.0: Your first LXD container [3/12] | Ubuntu [Electronic resource]. URL: https://ubuntu.com/blog/lxd-2-0-your-first-lxd-container (accessed: 11.12.2019).
139. Live Migration in LXD - LXD - system container manager [Electronic resource]. URL: https://lxd.readthedocs.io/en/latest/migration/ (accessed: 21.12.2019).
140. StatefulSets - Kubernetes [Electronic resource]. URL: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#limitations (accessed: 18.12.2019).
141. Virtuozzo Storage - OpenVZ Virtuozzo Containers Wiki [Electronic resource]. URL: https://wiki.openvz.org/Virtuozzo_Storage (accessed: 22.12.2019).
142. Apache Flink Stateful Functions 2.2 Documentation: Application Building Blocks [Electronic resource]. URL: https://ci.apache.org/projects/flink/flink-statefun-docs-stable/concepts/application-building-blocks.html#persisted-states (accessed: 14.03.2021).
143. Spark Streaming - Spark 3.1.1 Documentation [Electronic resource]. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html#caching--persistence (accessed: 14.03.2021).
144. Apache Kafka [Electronic resource]. URL: https://kafka.apache.org/25/documentation.html#api (accessed: 23.09.2021).
145. Streams Architecture — Confluent Documentation [Electronic resource]. URL: https://docs.confluent.io/platform/current/streams/architecture.html (accessed: 14.03.2021).
146. Apache Kafka Core Concepts [Electronic resource]. URL: https://kafka.apache.org/26/documentation/streams/core-concepts.
147. Apache Kafka’s MirrorMaker — Confluent Platform [Electronic resource]. URL: https://docs.confluent.io/4.0.0/multi-dc/mirrormaker.html (accessed: 15.04.2020).
148. Flink vs Kafka Streams - Comparing Features [Electronic resource]. URL: https://www.confluent.io/blog/apache-flink-apache-kafka-streams-comparison-guideline-users/ (accessed: 14.03.2021).
149. Redis [Electronic resource]. URL: https://redis.io/ (accessed: 01.02.2019).
150. Apache Storm [Electronic resource]. URL: https://storm.apache.org/ (accessed: 14.09.2019).

ПРИЛОЖЕНИЕ 1. Аббревиатуры

№	Аббре-виатура	Значение	Стра-ница
1	IoT	Интернет вещей (Internet of Things)	22
2	IIoT	Индустриальный интернет вещей (Industrial Internet of Things)	24
3	DT	Цифровой двойник (Digital Twin)	24
4	AV	Автономный автомобиль (Autonomous Vehicle)	26
5	EDA	Событийно-управляемая архитектура (Event-Driven Architecture)	29
6	SL	Слой хранения (Storage Layer)	38
7	PL	Слой обработки (Processing Layer)	38
8	DSPE	Платформы обработки потоков данных (Data Stream Processing Engines)	38
9	VM	Виртуальная машина (Virtual Machine)	41
10	DSS	Распределенная система хранения данных (Distributed Storage System)	44
11	API	Программный интерфейс приложения (Application Programming Interface)	45
13	SWF	Научный поток работ (Scientific Workflow)	48
14	DAG	Ориентированный ациклический граф (Directed acyclic graph)	50
15	SWfMS	Система управления научными потоками работ (Scientific Workflow Management System)	53
16	SDF	Синхронные потоки данных (Synchronous Data Flow)	55
17	PN	Сети процессов (Process Network)	55
18	PWM	Модель потока работ, поддерживающего разбиение (Partial Workflow Model)	57
19	CFD	Вычислительная гидродинамика (Computational Fluid Dynamics)	61
20	MWF	Микро-поток работ (Micro-Workflow)	65

ПРИЛОЖЕНИЕ 2. Основные обозначения

№	Обозначение	Значение	Страница
1.	W	Монолитный поток работ	68
2.	V	Множество вершин потока работ	68
3.	E	Множество ребер потока работ	68
4.	v_i	Вершина потока работ	68
5.	$deg^+(v_i)$	Выходная степень вершины, соответствующая количеству ребер, исходящих из v_i	68
6.	$deg^-(v_i)$	Входная степень вершины, соответствующая количеству ребер, направленных к v_i от других вершин	68
7.	(v_i, v_j)	Ребро, представляющее собой зависимость по данным от v_i к v_j	68
8.	S	Множество подпотоков работ	69
9.	V_i	Множество вершин, входящих в подпоток работ S_i	69
10.	E_i	Множество ребер между вершинами, входящими в V_i	69
11.	EI_i	Набор входных ребер с начальной вершиной вне подпотока работ S_i и конечной вершиной внутри подпотока работ S_i	69
12.	EO_i	Набор выходных ребер с начальной вершиной внутри подпотока работ S_i и конечной вершиной вне подпотока работ S_i	69
13.	VI_i	Набор вершин в S_i, расположенных на головной части ребер EI_i, а также вершин, не имеющих входных ребер	69
14.	VO_i	Набор вершин в S_i, расположенных на концах ребер EO_i, а также вершин, не имеющих выходных ребер	69
15.	cv_i	Вершина-потребитель	70

16.	pv_i	Вершина-генератор	70
17.	ECV_i	Набор ребер, идущих от cv_i к вершинам в VI_i	70
18.	EPV_i	Набор ребер, идущих от вершин в VO_i к cp_i	70
19.	MV_i	Множество всех вершин, находящихся внутри S_i, включая cv_i и pv_i	70
20.	ME_i	Множество всех ребер, расположенных внутри S_i, включая все ребра, которые идут из cv_i к вершинам в VI_i, а также все ребра, идущие от вершин в VO_i к cp_i	70
21.	MWF_i	Микро-поток работ из подпотока работ S_i	70
22.	Z	Матрицы представления монолитного потока работ W и множества подпотоков S, на которые разбивается поток работ W	73
23.	M_x	Матрица представления микро-потока работ MWF_x	75
24.	ts	Метка времени считывания данных	89
25.	$rcvTime$	Метка времени получения сообщения в системе обработки потоков работ	89
26.	mX_orgts	Метка времени, обозначающая момент, когда симулятор датчиков отправляет исходное сообщение номер X	115
27.	mX_rcvts	Метка времени, обозначающая момент, когда микро-поток работ получает исходное сообщение X из исходной темы Apache Kafka	115
28.	K_sentts	Метка времени, обозначающая момент, когда актер KafkaProducer отправляет итоговое сообщение в конечную тему Apache Kafka	115
29.	Av_SM	Средний интервал между исходными сообщениями	117
30.	Av_TAT	Среднее время обработки	117
31.	Av_L12	Средняя задержка	117