ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ПРОЦЕССЫ

УДК 004.6 Научная статья
DOI: 10.35330/1991-6639-2024-26-2-34-43
EDN: SLDLOB

Использование иерархических индексов для блокировки доступа к разделяемому ресурсу в микросервисах

В. С. Кириллов

Северо-Кавказский федеральный университет
355017, Россия, г. Ставрополь, ул. Пушкина, 1

Аннотация. В данной статье рассматривается инновационный алгоритм, предназначенный для эффективной блокировки доступа к разделяемым ресурсам в микросервисах. Главная особенность этого алгоритма заключается в унификации процессов блокировки и выполнения операций в многопоточных микросервисах с иерархической структурой разделяемых ресурсов. Использование данного алгоритма значительно упрощает разработку системы блокировки ресурсов и дает возможность изменять детализацию блокируемых ресурсов в соответствии с требованиями, предъявляемыми к обрабатываемым микросервисом сообщениям. Большим преимуществом алгоритма является возможность блокировки доступа к нескольким ресурсам без риска взаимной блокировки потоков. Это обеспечивает гарантию надежности и безопасности обработки сообщений микросервисами, особенно в случаях, когда необходимо обращаться к нескольким разделяемым ресурсам одновременно. Результаты исследования показывают, что предложенный алгоритм может значительно улучшить эффективность системы блокировки ресурсов в микросервисной архитектуре, снизить вероятность возникновения ошибок и упростить разработку программного обеспечения. В долгосрочной перспективе использование данного алгоритма может способствовать повышению производительности и надежности распределенных систем на основе микросервисов.

Ключевые слова: микросервисы, иерархии, разделяемые ресурсы

Поступила 01.03.2024, одобрена после рецензирования 26.03.2024, принята к публикации 02.04.2024

Для цитирования. Кириллов В. С. Использование иерархических индексов для блокировки доступа к разделяемому ресурсу в микросервисах // Известия Кабардино-Балкарского научного центра РАН. 2024. Т. 26. № 2. С. 34–43. DOI: 10.35330/1991-6639-2024-26-2-34-43

MSC: 68T027 Original article

Using hierarchical indexing for access control to shared resources in microservices

V.S. Kirillov

North Caucasus Federal University
355017, Russia, Stavropol, 1 Pushkin street

Abstract. This article discusses an innovative algorithm designed for efficient blocking of access to shared resources in microservices. The main feature of this algorithm lies in unifying the processes of resource blocking and operation execution in multithreaded microservices with a hierarchical structure of shared resources. The use of this algorithm significantly simplifies the development of a resource locking system and allows the customization of the level of detail in the blocked resources according to the

requirements imposed on the processed microservice messages. One major advantage of the algorithm is the ability to block access to multiple resources without the risk of thread deadlock. This ensures reliability and security in message processing by microservices, especially in cases where simultaneous access to multiple shared resources is required. The research results demonstrate that the proposed algorithm can significantly improve the efficiency of resource locking systems in a microservices architecture, reduce the likelihood of errors, and simplify software development. In the long term, the use of this algorithm can contribute to enhancing the performance and reliability of distributed systems based on microservices.

***Keywords**:* microservices, hierarchies, shared resources

Submitted 01.03.2024, approved after reviewing 26.03.2024, accepted for publication 02.04.2024

For citation. Kirillov V.S. Using hierarchical indexing for access control to shared resources in microservices. *News of the Kabardino-Balkarian Scientific Center of RAS.* 2024. Vol. 26. No. 2. Pp. 34–43. DOI: 10.35330/1991-6639-2024-26-2-34-43

Введение

С развитием разработки высоконагруженных систем произошла трансформация архитектуры серверной части. С приходом облачных вычислений и таких технологий, как Kubernetes [1, 2] и Docker [3], популярность микросервисов возросла. В настоящее время существует значительное число готовых фреймворков для построения микросервисов как от крупных компаний, так и open source решения. При проектировании микросервисов применяется ряд хорошо зарекомендовавших себя решений. В частности, широко используется удаленный вызов процедур. Данный подход появился сравнительно давно и претерпел ряд эволюционных изменений. Также широко известны решения, основанные на Corba [4], OLE/COM [5], однако данные решения не получили широкого распространения в проектировании микросервисов. Эти решения во многом зависят от операционной системы, в частности, ole/com решения возможно использовать только в windows-системах. К другим недостаткам данных технологий можно отнести сложность проектирования решений, использующих данные технологии. В микросервисах, основанных на удаленных вызовах процедур широко применяется gRPC [6] (решение, предложенное Google), использующее в качестве IDC определения сообщения Protobuf и Thrift [7] (решение, предложенное Meta).

Первое решение несколько более производительно, однако второе решение обладает более богатым функционалом. К недостаткам микросервисов, основанных на удаленных вызовах процедур, можно отнести синхронный вид взаимодействия между микросервисами. В корпоративном мире также получили достаточно большое распространение микросервисы, основанные на Restfull-технологии. Типичным примером такого подхода является решение, предлагаемое библиотекой Spring Boot [8], широко известной Java библиотеки Spring.

Данное решение в качестве метода кодирования сообщений использует Json-формат, и вызовы микросервисов происходят через Web-сервисы. К недостаткам такого подхода можно отнести большую latency и малую скорость обработки сообщений, а также зависимость от языка программирования (в данном случае Java). Другим подходом в программировании микросервисов является подход с асинхронными вызовами. Здесь есть две основные модели: архитектура без брокера сообщений и с ним. При проектировании микросервисной архитектуры с брокером сообщений (рис. 1) используются серверные решения на базе ActiveMQ, RabitMQ, Kafka и многие другие. Основная идея в данных системах строится на применении Publisher/Subscriber [12] шаблона проектирования.

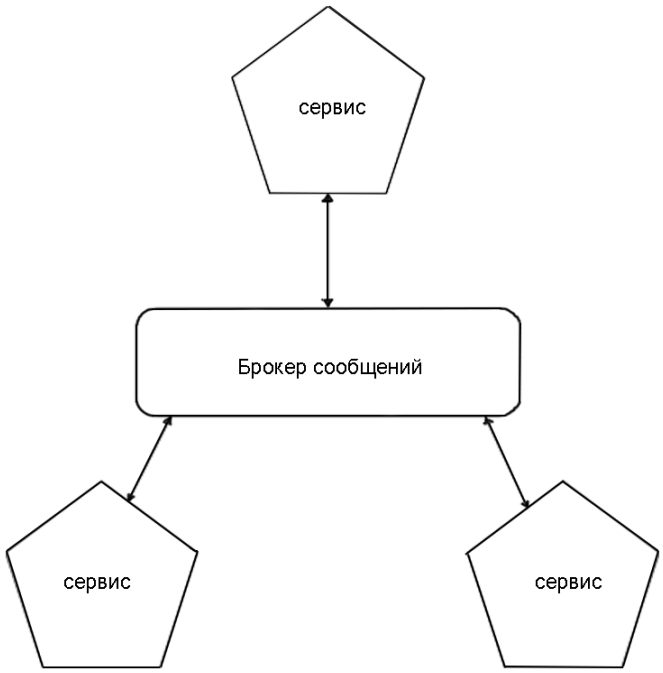

Рис. 1. *Микросервисная архитектура с брокером сообщений*

Fig. 1. *Microservice architeckture with message brocker*

Микросервис, который производит данные, публикует их в очередь на сервере. Микросервис, который обрабатывает данные, подписывается на них. При этом можно организовать чтение данных несколькими сервисами, а также выбрать точку, с которой начать обработку данных. Очень часто узким местом в такой архитектуре является брокер сообщений. К его производительности предъявляются большие требования. Системы, не использующие брокер сообщений (рис. 2), гораздо более гибкие.

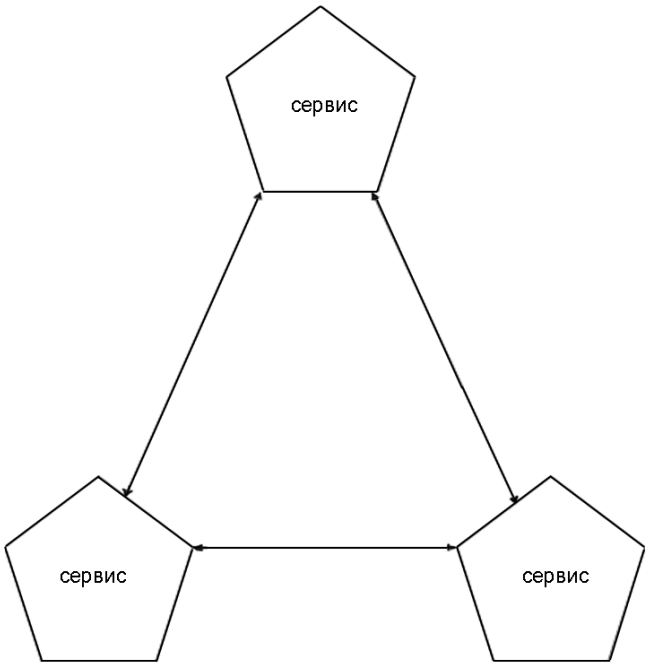

Рис. 2. *Микросервисная архитектура без брокера сообщений*

Fig. 2. *Microservice architeckture without message brocker*

В них возможно индивидуальное соединение микросервисов между собой, что избавляет систему от указанной выше проблемы. Однако проектирование таких систем представляет большие сложности, чем с брокером сообщений. Самыми значимыми библиотеками на данный момент для построения таких решений являются ZerroMQ и userver, но во многих компаниях существует достаточно много закрытых проприетарных решений.

При реализации асинхронного микросервиса часто применяются такие методы построения многопоточных приложений, как порождение отдельного потока исполнения (fork and join) [9] или использование очереди сообщений. Применение отдельных потоков исполнения вызывает проблемы при большом потоке сообщений, т.к. создание потока исполнения требует большого количества машинного времени. При реализации очереди сообщения попадают в очередь и поочередно выбираются потоками исполнения. Потоки исполнения при этом не создаются для каждого сообщения, а берутся из пула потоков. К настоящему времени создано достаточно большое количество примитивов для выполнения данного подхода.

Например, в библиотеках языка C# можно найти такую структуру, как BlockingCollection. Эта структура реализует концепцию producer/consumer [10, 11]. Нити, производящие данные в цикле, могут добавлять их в эту коллекцию. Нити, обрабатывающие данные в цикле, забирают их из коллекции. Эти операции выполнены потоко-безопасным образом. При отсутствии данных в коллекции нить-обработчик данных блокируется до того момента, как они не появятся в коллекции.

Количество записей в коллекции может быть ограничено, и в этом случае блокируется нить, добавляющая запись в коллекцию. Подобные структуры есть во многих других языках программирования. Этот подход хорошо работает, когда у нас решается задача, независимая по данным. В случае, когда для нас важен порядок выполнения операций и операции могут выполняться над одними и теми же данными, приходится дополнять микросервис средствами синхронизации. Например, если у нас реализуется микросервис, производящий зачисление и списание средств клиентов по счетам, а также расчет кредитоспособности клиента, то может возникнуть ситуация, когда вследствие того, что сообщения будут обработаны в случайном порядке, мы получим неправильный остаток на счете.

Более того, совершенно очевидно, что операции с разными валютами могут быть выполнены параллельно, и это не будет иметь взаимного воздействия, однако порядок операций с одной валютой должен быть соблюден. В этом случае можно произвести блокировку сообщений, основываясь на номере конкретного счета, и в случае, если система в настоящий момент обрабатывает сообщение, работающее с данным счетом, то следующее сообщение должно ожидать в очереди. Однако в случае, если нам нужно рассчитать объем средств, которыми владеет клиент для работы на фондовом рынке, то нужно произвести блокировку всех счетов клиента.

Также при конвертации средств из одной валюты в другую нужно блокировать два счета. Совершенно очевидно, что необходим универсальный метод решения данной проблемы, так как решение этой задачи путем создания критических секций для каждой отдельной проблемы трудоемко. В нашей работе предлагается универсальный механизм обработки сообщений, которые могут иметь зависимость по данным. Метод позволяет писать обработчик сообщения без произведения дополнительной синхронизации.

Теоретический подход

Рассмотрим поток различных сообщений, приходящих в микросервис. Данные сообщения относятся к одной доменной области, однако при этом должны быть обработаны различным образом. Возможна ситуация, когда порядок обработки сообщений может влиять на результат. Рассмотрим последовательность сообщений, представленных на рис. 3. В данной последовательности мы предположили, что сообщения x_1, x_2, x_5 влияют на состояние некоторого разделяемого ресурса X_1, x_3 на X_2, а x_4 на X_3 и X_2, причем X_1 является родительским ресурсом для X_2 и X_3. Сообщения y_1 на Y_1, y_2, y_3 на Y_2, а y_4 на Y_3, причем состояние Y_1 является родительским для Y_2 и Y_3. Сообщение xy изменяет состояния X_1 и Y_1.

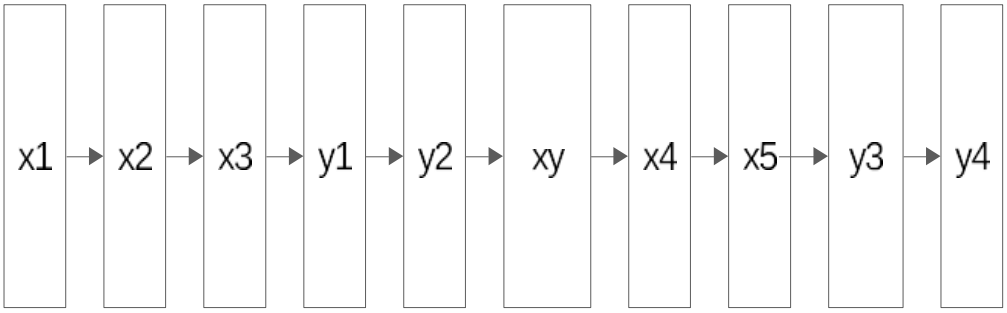

Рис. 3. *Последовательность сообщений*

Fig. 3. *Sequence of messages*

Предполагаемый порядок обработки сообщений представлен на рис. 4. Здесь сообщения, влияющие на некоторый разделяемый ресурс, выполняются в порядке доставки в микросервис, однако порядок обработки сообщений x и y не гарантируется. В приведенном примере сообщение x_1 заблокировало выполнение сообщений x_2, x_3, потому как состояние X_1 является родительским для X_2 и X_3. В последовательности сообщений есть сообщения xy, которые могут влиять также на состояния ресурсов X_1 и Y_1. В этом случае обработка сообщений x_i и y_i должна быть приостановлена при достижении сообщения xy и продолжена только после того, как это сообщение будет обработано.

Данный алгоритм достаточно трудно реализовать, основываясь на традиционных примитивах синхронизации [13, 14] (мьютексы, семафоры и т.д.), т.к. в общем случае мы не знаем количество разделяемых ресурсов, а также время обработки каждого сообщения может быть различным вследствие того, что для выполнения действий нам, возможно, понадобятся дополнительные данные и мы вынуждены будем сделать вызов в другой микросервис для их получения. Обработка этих сообщений в одном потоке исполнения также не выглядит перспективной, потому как вызов в другой микросервис достаточно часто занимает много машинного времени и исполнение всех остальных сообщений будет приостановлено.

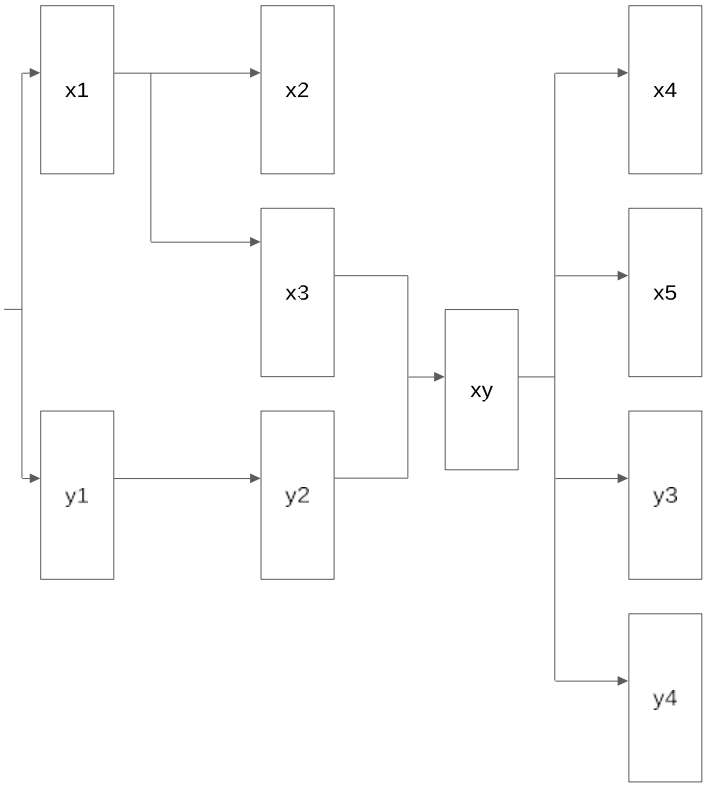

Рис. 4. *Обработка сообщений*

Fig. 4. *Messages processing*

Рассмотрим алгоритм, представленный на рис. 5.

Сообщения при доставке в микросервис попадают во входную очередь сообщений, и таким образом достигается асинхронность работы данного микросервиса. По запросу обрабатывающего потока мы берем первое сообщение в очереди на обработку. Для выбранного сообщения генерируется необходимое количество блокирующих ключей на основании типа сообщения и содержания сообщения. Данные ключи представляют собой структуры, описывающие иерархию разделяемых ресурсов и позволяющие быстро определять, является ли один из ключей потомком или предком другого. Примеры генерации таких ключей можно найти в следующих источниках[1, 2, 3] [15].

Помимо очереди сообщений, предлагаемый алгоритм использует список ключей и список блокированных ключей. Ключи, определяющие ресурсы, блокированные сообщениями, которые в данный момент обрабатываются многопоточной системой, помещаются в список ключей. В очереди сообщений находится сообщение, являющееся претендентом на обработку. Когда у нас появляется очередной свободный поток, он просматривает голову очереди сообщений и генерирует ключи для претендента на обработку. Далее мы проверяем, находятся ли в списке ключей предки или потомки данного ключа. Если они там находятся, то ключ помещается в список блокированных ключей. Несмотря на

[1]Tropashko V. Trees in SQL: Nested Sets and Materialized Path. [Электронный ресурс]. 2002. URL: www.dbazine.com/tropashko4.shtml.

[2]Tropashko V. Nested Intervals with Farey Fractions. [Электронный ресурс]. 2004. URL: arXiv preprint cs.DB/0401014.

[3] PostgreSQL index – ltree [Электронный ресурс]. URL: http://www.sai.msu.su.

то, что рассматривается только одно сообщение в качестве претендента на обработку, мы используем список блокированных ключей, т.к. сообщения могут генерировать более одного ключа. В случае, если нет ни одного ресурса, который уже используется и необходим для обработки данного сообщения, ключи сообщения заносятся в список ключей, и сообщение обрабатывается. Если же конфликт доступа к ресурсам обнаружен, то поток будет находиться в состоянии ожидания. После окончания обработки сообщения все связанные с этим сообщением ключи удаляются из списка ключей и списка блокированных ключей. Если в списке блокированных ключей нет больше записей, значит, следующее сообщение готово к обработке. В алгоритме присутствуют две области кода, которые должны выполняться атомарно. Первая область начинается с просмотра сообщения в очереди и заканчивается перед обработкой сообщения. Вторая начинается после обработки сообщения и заканчивается проверкой блокированных ключей. Таким образом, только обработка сообщения выполняется многопоточно.

Рис. 5. *Алгоритм обработки сообщений*

Fig. 5. *Algorithm of message processing*

Анализ метода

Рассмотрим порядок прохождения сообщений, приведенных на рис. 3. Предлагаемый алгоритм не просматривает все сообщения, а только проверяет возможность параллельного исполнения следующего сообщения. Вследствие этого нам не удалось добиться оптимального порядка обработки сообщений, и у нас будет субоптимальный порядок (рис. 6). Произведем оценку сложности по времени работы алгоритма.

Добавление записи в очередь и чтение первой записи выполняется за время O(1). Каждое сообщение генерирует небольшое количество ключей (на практике не больше 4–5), вследствие этого список блокированных ключей можно реализовать при помощи списка, и поиск, удаление блокированного ключа будет иметь сложность O(n), где n – количество ключей, генерируемое для каждого сообщения, а добавление ключа – O(1).

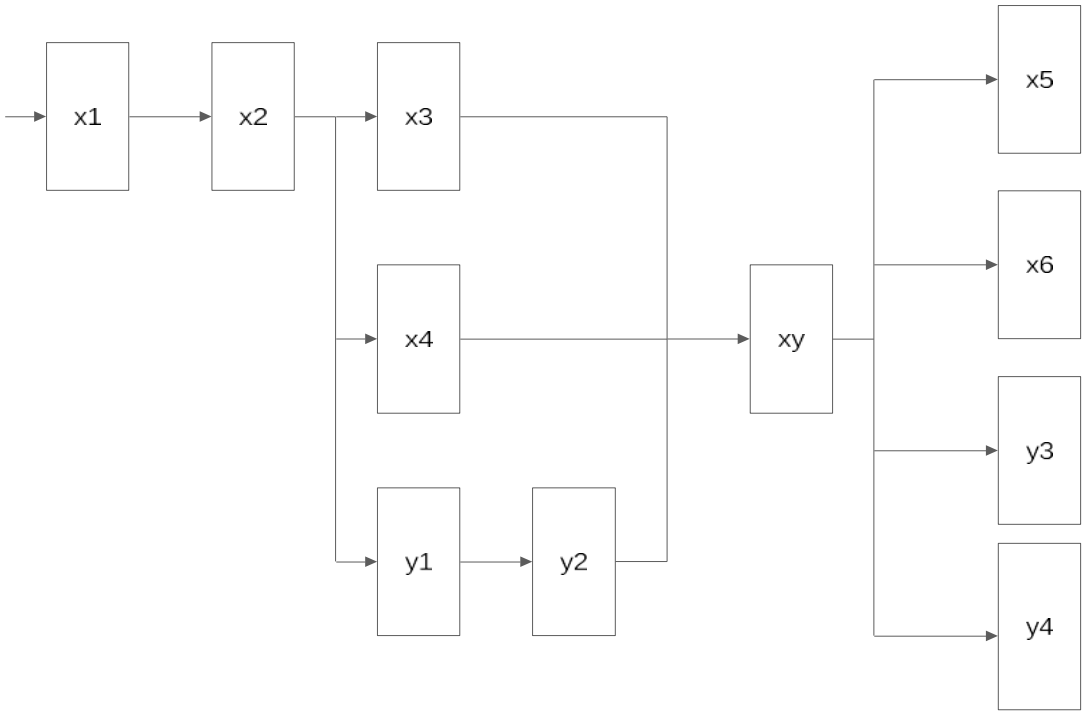

Рис. 6. *Субоптимальный порядок обработки сообщений*

Fig. 6. *Suboptimal message processing*

Список ключей, напротив, может быть достаточно обширен при большом параллелизме системы, и его можно реализовать при помощи упорядоченного множества (ordered set). В этом случае операции добавления, поиска и удаления записи выполняются за O(log m), где m – количество ключей обрабатываемых сообщений.

Рассмотрим возможность взаимной блокировки сообщений с несколькими ключами. Как мы видим, алгоритм имеет 2 атомарные секции, однако они имеют доступ к одному и тому же разделяемому ресурсу – списку ключей. Следовательно, это реализуемо при помощи одного объекта синхронизации (например, mutex). При этом происходит единовременная проверка возможности заблокировать доступ к некоторым ресурсам, и, следовательно, невозможна ситуация, когда 2 сообщения потребуют доступ к одним и тем же ресурсам и будут одновременно блокировать доступ в зеркальном порядке. Следовательно, взаимная блокировка невозможна.

Заключение

Приведенный алгоритм позволяет изменять детализацию блокируемых ресурсов, при этом не увеличивая сложности написания процедур обработки сообщений, создавать иерархические структуры блокируемых ресурсов, что позволяет выбирать, на каком уровне иерархии мы планируем сделать блокировку ресурса. Представляемый подход позволяет программисту декларировать, какие ресурсы необходимы для обработки, и не оперировать далее категориями многопоточного исполнения. Процедуры обработки сообщений пишутся исходя из парадигмы однопоточной среды, что в значительной мере уменьшает возможность внесения в код ошибок, связанных с многопоточностью. Алгоритм лишен такой проблемы, как взаимная блокировка ресурсов, что также положительным образом влияет на отказоустойчивость систем, он достаточно прост в исполнении, однако при этом предлагает неоптимальную последовательность обработки сообщений.

К недостаткам также можно отнести то, что блокировка ресурса происходит перед началом обработки сообщения, а снятие блокировки – после. Применение данного алгоритма положительно сказывается на увеличении отказоустойчивости микросервисов.

REFERENCES

1. Lukˇsa M. Kubernetes in Action. NY: Manning Publications Co., 2018. 624 p.

2. Rosso J., Lander R., Brand A., Harris J. Production Kubernetes. Sebastopol, California: O'Reilly Media, Inc., 2021. 508 p.

3. Nickoloff J., Kuenzli S. Docker in Action, 2 edition. NY: Manning Publications Co., 2019. 336 p.

4. Brose G., Vogel A., Dubby K. Java programming with CORBA: advanced techniques for building destributed applications. USA: Wiley Computer Publishing, 2001. 710 p.

5. Brose G., Vogel A., Dubby K. OLE automation programmer's reference: creating programmable 32-bit applications. USA: Redmond, Wash.: Microsoft Press, 1996. 399 p.

6. Humphries J., Konsumer D., Muto D. Practical gRPC. USA: Bleeding Edge Press, 2018. 169 p.

7. Abernethy R.Programmer's Guide to Apache Thrift. NY: Manning Publications Co., 2019. 592 p.

8. Walls C. Spring Boot in Action. NY: Manning Publications Co., 2015. 264 p.

9. Andrews G.R., Schneider F.B. Concepts and Notations for Concurrent Programming. Computing Surveys. Vol. 15. No. 1. 1983. Pp. 3–43. https://dl.acm.org/doi/pdf/10.1145/356901.356903

10. Wirth N. Toward a discipline of real-time programming. Comm. of the ACM. Vol. 20. No. 8. 1977. Pp. 577–583. https://dl.acm.org/doi/pdf/10.1145/359763.359798

11. Kevin J. The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems. *SAC '93:* Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing, 1993.

12. Ozansoy C., Zayegh A., Kalam A. The Real-Time Publisher/Subscriber Communication Model for Distributed Substation Systems. IEEE Transactions on Power Delivery. Vol. 22(3). 2007. Pp. 1411–1423.

13. Silberschatz A., Gagne G., Galvin B.P. Operating System Concepts. NY: John Wiley & Sons, 2008. 971 p.

14. Hennessy John L., Patterson David A. Computer Architecture: A Quantitative Approach: Morgan Kaufmann, 2011. 476 p.

15. O'Neil P., O'Neil E., Pal Sh. et al. ORDPATHs: Insert-Friendly XML Node Labels. Proceedings of the 2004 ACM SIGMOD international conference on Management of data. 2004. Pp. 903–908.

Информация об авторе

Кириллов Владимир Святославович, канд. физ.-мат. наук, доцент, Северо-Кавказский федеральный университет;
355017, Россия, г. Ставрополь, ул. Пушкина, 1;
vkirillov74@gmail.com, ORCID: https://orcid.org/0009-0007-3996-1844

Information about the author

Vladimir S. Kirillov, Candidate of Physical and Mathematical Sciences, North Caucasus Federal University;
355017, Russia, Stavropol, 1 Pushkin street;
vkirillov74@gmail.com, ORCID: https://orcid.org/0009-0007-3996-1844