ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

С.А. Гаранин

МОДЕЛИ И ТЕХНОЛОГИИ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ

Учебно-методическое пособие
по выполнению лабораторных работ

для студентов
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2025

УДК 004.75:519.6
ББК 6Ф7.3
Г20

Рецензент:
Феоктистова О.Г. – д-р техн. наук, профессор

Гаранин С.А.
Г20 Модели и технологии распределенных вычислений [Текст] : учебно-методическое пособие по выполнению лабораторных работ / С.А. Гаранин. – М.: ИД Академии Жуковского, 2025. – 32 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Модели и технологии распределенных вычислений» по учебному плану для студентов направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 28.01.2025 г. и методического совета 28.01.2025 г.

УДК 004.75:519.6
ББК 6Ф7.3

В авторской редакции

Подписано в печать 23.05.2025 г.
Формат 60x84/16 Печ. л. 2 Усл. печ. л. 1,86
Заказ № 1082/0325-УМП18 Тираж 25 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (499) 755-55-43
E-mail: zakaz@itsbook.ru

СОДЕРЖАНИЕ

1. Введение

Современный этап развития информационных технологий характеризуется стремительным ростом объемов данных, усложнением вычислительных задач и необходимостью обработки информации в режиме реального времени. В этих условиях традиционные централизованные системы вычислений зачастую оказываются недостаточно эффективными, что приводит к активному развитию и внедрению распределенных вычислительных систем.

Распределенные вычисления позволяют объединять ресурсы множества устройств, расположенных на значительном расстоянии друг от друга, для решения сложных задач, требующих высокой производительности и отказоустойчивости.

Дисциплина "Модели и технологии распределенных вычислений" посвящена изучению принципов организации, методов проектирования и технологий реализации распределенных систем. Она охватывает широкий спектр вопросов, начиная от теоретических основ распределенных вычислений, таких как модели взаимодействия процессов, синхронизация и согласованность данных, до практических аспектов, включая использование современных технологий и платформ, таких как облачные вычисления, грид-системы, параллельные вычисления и блокчейн.

Целью данного учебно-методического пособия является формирование у студентов системного понимания принципов работы распределенных систем, а также развитие навыков проектирования и реализации распределенных приложений.

Пособие необходимо студентам на всех этапах, начиная от подготовки до оформления отчета и защиты лабораторной работы.

2. Организационно-методические рекомендации

В соответствии с учебным планом подготовки студентов по направлению 09.03.01 «Информатика и вычислительная техника», направленность «Интеллектуальные системы обработки и анализа данных» и рабочей программой по дисциплине «Модели и технологии распределенных вычислений» и изложенными в них требованиями к уровню подготовки для работы в организациях гражданской авиации, студенты должны обладать навыками:

- владения современными инструментами разработки и отладки программного обеспечения;
- использования инструментальных средств для извлечения, преобразования, хранения и обработки данных из разнородных источников, в том числе в режиме реального времени;
- программирования на языках высокого уровня.

В соответствии с учебной программой продолжительность лабораторных работ – 8 часов (2 лабораторные работы по 4 часа).

2.1. Компетенции обучающегося, формируемые в результате освоения дисциплины «Модели и технологии распределенных вычислений»

В результате изучения дисциплины студенты должны знать:

- Способы решения вычислительных задач с применением распределенных вычислений (ПК-1.1.12);

уметь:

- Формировать и программировать алгоритмы распределенных вычислений в предметной среде (ПК-1.2.11);
- Разрабатывать модели распределенных вычислений для аналитики больших данных (ПК-1.3.9).

Дисциплина «Модели и технологии распределенных вычислений» направлена на обеспечение студентов необходимыми знаниями, умениями и навыками при освоении следующих дисциплин:

- Разработка профессиональных приложений;
- Моделирование вычислительных систем и сетей;
- Программирование сетевых приложений;
- Методы визуального и параллельного программирования;
- Системы искусственного интеллекта;
- Мультиагентное моделирование;
- Автоматизированные системы ОрВД;
- Автоматизированные системы организации и управления на ВТ.

2.2. Этапы выполнения лабораторных работ

Лабораторная работа представляет собой структурированный процесс, состоящий из нескольких последовательных этапов, каждый из которых направлен на достижение определенных учебных целей. К основным этапам выполнения лабораторной работы относятся:

1. Домашняя подготовка. На данном этапе студент осуществляет предварительную работу, которая включает:
   - Изучение теоретического материала, представленного в лекциях, для формирования понимания ключевых концепций и принципов, необходимых для выполнения работы.
   - Тщательное ознакомление с заданием и индивидуальным вариантом, что позволяет четко определить цели, задачи и требования к работе.
   - Подготовку необходимых файлов с исходными данными, если это предусмотрено заданием.
   - Начальное оформление отчета, включающее описание целей, задач и предполагаемых методов выполнения работы.
2. Выполнение работы в соответствии с вариантом задания. Этот этап осуществляется в ходе занятий в компьютерном классе МГТУ ГА под руководством преподавателя. Студент выполняет практическую часть работы,

применяя полученные теоретические знания и используя предоставленные инструменты и ресурсы.

3. Сдача выполненной работы преподавателю. После выполнения всех практических задач студент представляет результаты своей работы преподавателю для предварительной проверки и, при необходимости, вносит корректировки в соответствии с полученными замечаниями. На этом этапе преподаватель оценивает корректность выполнения задания, соответствие результатов поставленным требованиям и качество их оформления.

4. Оформление отчета. Отчет является обязательной частью лабораторной работы и должен содержать подробное описание всех этапов выполнения задания, включая цели, методы, исходные данные, полученные результаты и выводы. Отчет оформляется в соответствии с установленными стандартами и требованиями.

5. Защита лабораторной работы. Завершающим этапом является защита работы, в ходе которой студент представляет отчет и отвечает на вопросы преподавателя, касающиеся как практической части работы, так и теоретического материала по теме исследования.

Зачет по лабораторной работе выставляется преподавателем на основании качества выполнения задания, правильности оформления отчета и уровня понимания студентом теоретических основ.

2.3. Отчет по лабораторной работе

По завершении выполнения лабораторной работы студент обязан оформить отчет, который должен содержать следующие обязательные элементы:

- название лабораторной работы;
- цель работы;
- задание на выполнение работы;
- индивидуальный вариант работы;
- последовательное описание выполнения работы, сопровождаемое подробными пояснениями (комментариями) к каждому этапу;
- результаты выполнения работы, сопровождаемые анализом и комментариями, которые демонстрируют достижение поставленных целей и соответствие полученных данных ожидаемым результатам;
- выводы по проделанной работе, в которых студент обобщает полученные результаты, оценивает их значимость и формулирует основные заключения.

В отчете должны быть представлены скриншоты, подтверждающие выполнение каждого шага, а также промежуточные результаты. Скриншоты должны быть выполнены не для всего экрана, а только для его фрагмента, содержащего актуальную информацию. Размер шрифта в распечатанном отчете должен обеспечивать удобство чтения. Рекомендуется использовать шрифт Times New Roman размером 14 пунктов.

Если в отчет включены рисунки, выполненные вручную или представленные в виде фотографий, они должны быть выполнены аккуратно, с обязательным использованием линейки для обеспечения четкости и читаемости.

Титульный лист отчета должен содержать название дисциплины, фамилию и номер группы студента. Также на титульном листе указываются полное название вуза, наименование кафедры и фамилия преподавателя, курирующего выполнение работы. Форма титульного листа приведена в Приложении 1.

Соблюдение указанных требований к оформлению отчета является обязательным и способствует структурированному представлению результатов выполненной работы, а также облегчает процесс проверки и оценки со стороны преподавателя.

2.4. Защита лабораторной работы

После завершения выполнения учебного задания и подготовки соответствующего отчета студент представляет результаты своей работы на защите. Защита работы является важным этапом учебного процесса, в ходе которого преподаватель осуществляет комплексную проверку представленных материалов. Оценивается правильность оформления отчета, включая соблюдение установленных стандартов и требований к структуре, содержанию и оформлению документации.

Далее преподаватель анализирует соответствие выполненной работы исходному заданию, проверяя, насколько полно и точно были реализованы поставленные цели и задачи. Особое внимание уделяется корректности использования исходных данных, а также обоснованности полученных результатов и их соответствию проделанной работе.

В ходе защиты студенту предлагается ответить на вопросы преподавателя, которые могут касаться как практической части работы, включая методы и подходы, использованные при выполнении задания, так и теоретических аспектов, связанных с темой работы. Это позволяет оценить не только уровень практических навыков студента, но и глубину его теоретической подготовки, понимание ключевых концепций и способность аргументированно обосновывать принятые решения.

3. Лабораторная работа №1

«Методы реализации векторных часов»

Продолжительность выполнения работы – 4 часа.

3.1. Цель работы

Целью лабораторной работы является приобретение навыков работы с концепцией векторных часов и методами их реализации в распределенных системах для обеспечения непротиворечивости и упорядочивания событий, а

также закрепление понимания причинно-следственного порядка в распределенных системах.

3.2. Краткие теоретические сведения

Механизм векторных часов является ключевым инструментом для отслеживания причинно-следственных зависимостей между событиями в распределённых системах.

В отличие от скалярных часов, которые фиксируют одно значение времени для всей системы, векторные часы предоставляют каждому процессу в распределённой системе собственные локальные часы, что позволяет отслеживать ход выполнения всех процессов и их взаимодействие.

Система векторных часов формирует частичный порядок на множестве событий, обеспечивая строгую непротиворечивость и возможность выявления параллельных событий.

Описание векторных часов

Векторное время представляется в виде N-мерного вектора, где N — количество процессов в распределённой системе. Вектор Vi процесса Pi содержит текущее состояние логических часов этого процесса. То есть Vi=[Vi[1],Vi[2],…,Vi[N]], где каждый компонент Vi[k] представляет собой локальное время процесса Pi, отражающее количество событий, произошедших в процессе Pi.

Отметка времени события ei, происходящего в процессе Pi, обозначается как векторное время V(ei), что соответствует текущему состоянию вектора Vi на момент наступления события.

Правила продвижения векторных часов

Для соблюдения непротиворечивости векторных часов в системе применяются следующие правила:

Правило 1: Локальное продвижение времени

Перед выполнением любого события процесс Pi увеличивает свои локальные часы:

$$Vi[i]=Vi[i]+d,\ d>0$$

Обычно d принимается равным 1, что означает, что перед каждым событием процесс увеличивает значение своего локального времени на единицу.

Правило 2: Обработка полученного сообщения

Когда процесс Pj получает сообщение с векторной отметкой времени Vmsg от процесса Pi, процесс Pj обновляет свои часы и выполняет следующие действия:

1. Обновляет свой вектор времени, синхронизируя его с вектором времени полученного сообщения:

$$Vj[k]=\max(Vj[k],Vmsg[k]),\ 1\leq k\leq N$$

2. Выполняет продвижение локальных часов по Правилу 1:

$$Vj[j]=Vj[j]+1$$

3. Обрабатывает полученное сообщение.

Операции сравнения векторных отметок времени

Для того чтобы понять причинно-следственные связи между событиями, необходимо ввести операции сравнения между векторными отметками времени.

Пусть V(ei) и V(ej) — это векторные отметки времени двух событий ei и ej. Определим следующие отношения:

V(ei)=V(ej) — векторные отметки времени равны, если для всех компонентов k:

$$Vi[k]=Vj[k],\ \forall k$$

V(ei)≤V(ej) — векторное время события ei не опережает векторное время события ej, если для всех компонентов k выполняется неравенство:

$$Vi[k]\leq Vj[k],\ \forall k$$

V(ei)<V(ej) — векторное время события ei строго меньше векторного времени события ej, если для всех компонентов k выполняется:

$$Vi[k]\leq Vj[k],\ \forall k,\ \exists k{:}Vi[k]<Vj[k]$$

Если V(ei) и V(ej) не удовлетворяют ни одному из этих условий, то такие события называются параллельными, и их векторные отметки времени считаются несравнимыми:

$$V(ei)||V(ej) \Leftrightarrow \neg(V(ei)<V(ej))\wedge\neg(V(ej)<V(ei))$$

Строгая непротиворечивость векторных часов

Механизм векторных часов обеспечивает строгую непротиворечивость, что позволяет точно отслеживать причинно-следственные зависимости между событиями.

Это значит, что для любых двух событий ei и ej, если одно событие причинно влияет на другое, то их векторные отметки времени подчиняются отношению:

$$ei\rightarrow ej \Leftrightarrow V(ei)<V(ej)$$

Кроме того, если два события параллельны, то их векторные отметки времени будут несравнимы:

$$ei||ej \Leftrightarrow V(ei)||V(ej)$$

Это демонстрирует, что векторные часы сохраняют причинно-следственные связи между событиями распределённой системы.

Пример использования векторных часов

Рассмотрим пример распределённой банковской системы, где несколько процессов P1,P2,…,PN взаимодействуют для подсчета общей суммы на счетах. Каждый процесс ведет учёт суммы в своём локальном хранилище, и между ними обмениваются сообщениями. Для того чтобы все процессы согласовали общий результат, они могут использовать векторные часы.

Процесс Pi может в определённый момент времени вычислить свой вектор времени Vi и послать его всем остальным процессам. Каждый процесс, получив векторное время от других процессов, обновляет своё время, соблюдая правила продвижения и обновления локальных часов.

В процессе передачи сообщений и синхронизации векторных часов можно отслеживать, какие события происходят до других, и гарантировать, что в итоге все процессы согласуют свои данные без ошибок, избегая конфликтов и некорректных обновлений.

Методы эффективной реализации векторных часов

Размер векторных отметок времени определяется числом N процессов, участвующих в распределенном вычислении, и растет линейно с увеличением N. В случае, когда число процессов велико, использование механизма векторных часов будет приводить к необходимости пересылки значительного объема данных с каждым сообщением.

Поэтому если в распределенной системе работают сотни процессов даже при небольшом числе событий, происходящих всего в нескольких процессах, доля накладных расходов в каждом пересылаемом сообщении может достигать недопустимых величин. Несмотря на то, что для выполнения условия строгой непротиворечивости длина векторных отметок времени не может быть меньше числа процессов в распределенной системе, существует ряд методов, позволяющих реализовать механизм векторных часов более эффективно.

Метод дифференциальной пересылки векторного времени

Метод дифференциальной пересылки векторного времени основан на том наблюдении, что между последовательными отправками сообщений одному и тому же получателю обычно изменяется лишь небольшое число компонентов векторного времени процесса-отправителя.

Это становиться особенно заметным, когда распределенная система состоит из большого числа процессов, но информацией друг с другом активно обмениваются лишь немногие из них. Поэтому, когда процесс Pi отправляет сообщение процессу Pj, предлагается вместе с этим сообщением передавать

только те компоненты векторного времени процесса Pi, которые изменились с момента последней отправки сообщения процессу Pj.

Таким образом, если с момента последней отправки сообщения процессу Pj в процессе Pi изменились только компоненты i1, i2, … in его векторного времени Vi на значения соответственно v1, v2, … vn, то в новом сообщении процесс Pi пересылает лишь множество пар: {(i1, v1), (i2, v2), … (in, vn)}.

Когда процесс Pj получает такое сообщение, он обновляет свое векторное время согласно следующему правилу:

$$Vj[ik] = \max(Vj[ik], vk),\ 1 \le k \le n.$$

Данный метод позволяет уменьшить накладные расходы на передачу сообщений и, как следствие, снизить требования к емкости каналов связи и размеру буфера приема-передачи на отправляющей и принимающей стороне.

Конечно, в наихудшем случае процессу придется передавать все компоненты своего векторного времени, однако, в среднем можно ожидать, что размер такой дифференциальной отметки времени, передаваемой с каждым сообщением, будет меньше, чем размер всего вектора Vi.

Стоит обратить внимание, что для правильной работы этого метода каналы должны обладать свойством FIFO. Метод дифференциальной пересылки векторного времени требует, чтобы каждый процесс хранил значения всех своих отметок времени для последних событий отправки сообщений каждому другому процессу в распределенной системе.

Прямая реализация данного требования приводит к накладным расходам на хранение отметок времени в каждом процессе с порядком роста O(N2). Однако существует метод, позволяющий уменьшить накладные расходы, связанные с хранением дополнительной информации в каждом процессе, до O(N).

Суть его заключается в следующем. Каждый процесс Pi дополнительно поддерживает работу с двумя массивами:

- LSi[1..N] (от англ. Last Sent). Элемент LSi[j] содержит значение логического локального времени процесса Pi, Vi[i] на момент последней отправки процессом Pi сообщения процессу Pj.
- LUi[1..N] (от англ. Last Update). Элемент LUi[j] содержит значение логического локального времени процесса Pi, Vi[i] на момент последнего изменения процессом Pi j-го компонента своего векторного времени Vi[j].

Очевидно, LUi[i] = Vi[i] в каждый момент выполнения процесса Pi, а значение LUi[j] обновляется при получении процессом Pi сообщения, вызывающего изменение Vi[j]. В свою очередь значение элемента LSi[j] обновляется каждый раз при отправке процессом Pi сообщения процессу Pj.

Используя эти два массива, можно заключить, что с момента последней отправки процессом Pi сообщения процессу Pj в процессе Pi изменились только те k компонентов его векторного времени Vi[k], для которых LSi[j] < LUi[k].

Поэтому, согласно методу дифференциальной пересылки векторного времени, только эти компоненты должны быть переданы в новом сообщении из

Pi в Pj. Таким образом, когда процесс Pi отправляет сообщение процессу Pj, вместе с этим сообщением в качестве отметки времени он пересылает лишь множество пар

$$\{(k, Vi[k]): LSi[j] < LUi[k]\},$$

вместо пересылки всего вектора Vi, состоящего из N компонентов.

Пример работы метода дифференциальной пересылки векторных отметок времени приведен на рис. 1. На этом рисунке второе сообщение, переданное процессом P3 в процесс P2 с отметкой времени {(3,2)}, информирует процесс P2 о том, что третий компонент векторного времени был изменен, и его новое значение равно двум. Передаваемая отметка времени выглядит именно так, потому что с момента последней отправки сообщения из P3 в P2 процесс P3 изменил показания своих локальных часов с единицы до двух.

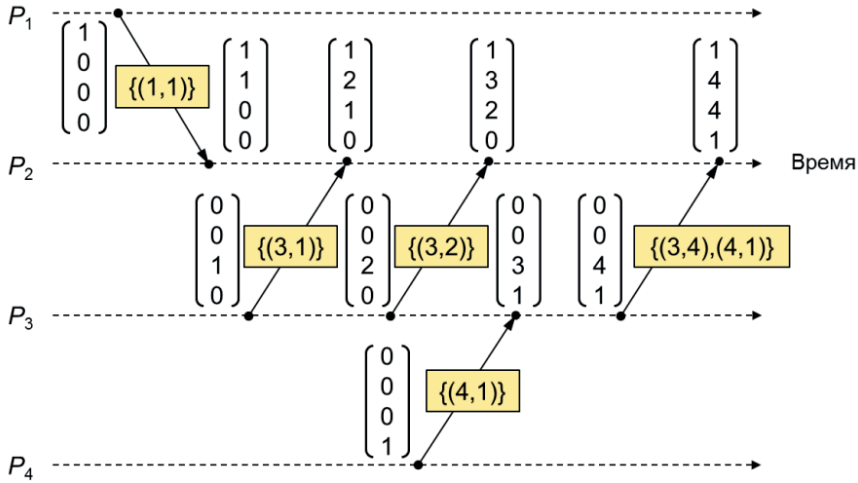

Рис. 1. Пример работы метода дифференциальной пересылки векторного времени.

Таким образом можно сделать вывод, что применение представленного метода дифференциальной пересылки векторного времени может привести к уменьшению накладных расходов, связанных с передачей сообщений, особенно, в случае, когда взаимодействие процессов в распределенной системе носит локальный характер.

Механизм логических часов с фиксацией только прямых зависимостей

Механизм логических часов, фиксирующих только прямую зависимость между процессами, позволяет снизить накладные расходы на передачу сообщений путем пересылки всего одной скалярной величины с каждым

сообщением. В этой связи "полноценное" векторное время, позволяющее отслеживать причинно-следственный порядок событий распределенного вычисления, становится недоступным для процессов непосредственно во время их выполнения, т.е. в режиме on-line. Вместо этого, каждый процесс поддерживает лишь информацию, позволяющую выявлять только прямую зависимость его событий от событий в других процессах. При этом говорят, что событие ej процесса Pj находится в прямой зависимости от события ei другого процесса Pi, если между наступлением событий ei и ej состоялась прямая передача сообщения из Pi в Pj.

В механизме логических часов, фиксирующих прямую зависимость, каждый процесс Pi вместо вектора Vi поддерживает работу с так называемым вектором прямых зависимостей (англ. direct dependency vector) Di[1..N]. В компоненте Di[k] этого вектора хранится локальное время процесса Pk на момент отправки последнего сообщения из Pk в Pi.

Пример работы часов, фиксирующих прямую зависимость, приведен на рис. 2. На этом рисунке, когда процесс P4 отправляет сообщение процессу P3, вместе с этим сообщением он передает только значение своего логического времени, равное двум, что позволяет отследить прямую зависимость второго события e32 процесса P3 от второго события e42 процесса P4. Затем процесс P3 отправляет сообщение в процесс P2, тем самым устанавливая прямую зависимость P2 от P3.

Таким образом, в действительности, четвертое событие e24 процесса P2 становится зависимым от второго события e42 процесса P4 согласно отношению причинно-следственного порядка. Однако, процесс P2 не получал об этом никакой информации и непосредственно по ходу своего выполнения установить эту зависимость не может. Более того, в четвертом элементе вектора зависимостей процесса P2, D2[4], содержится информация о более раннем событии e41 процесса P4, от которого e24 зависит напрямую.

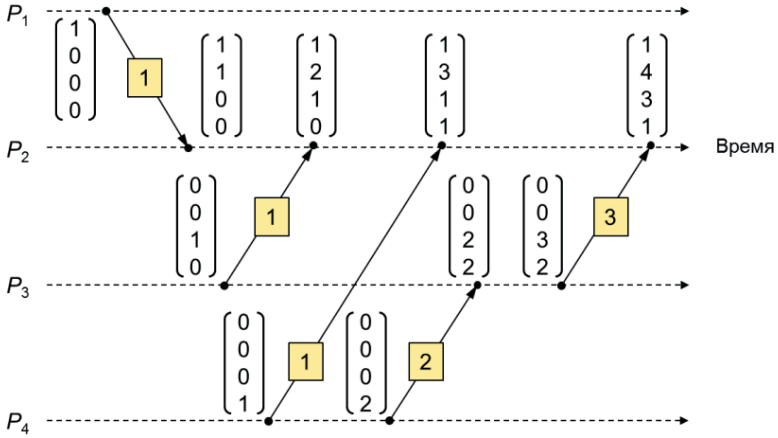

Рис. 2. Пример работы часов, фиксирующих прямую зависимость.

Этот пример наглядно демонстрирует тот факт, что рассматриваемый механизм логических часов сам по себе не сохраняет транзитивные зависимости причинно-следственной связи между событиями. Такие зависимости могут быть определены только в режиме off-line с помощью рекурсивного поиска на множестве векторов прямых зависимостей, зафиксированных процессами для своих событий.

Очевидно, такой поиск приводит к дополнительным вычислительным затратам и задержкам определения векторного времени того или иного события. Поэтому данный механизм логических часов применяется только для решения таких задач, которые не требуют выявления транзитивной причинно-следственной зависимости между событиями непосредственно "на лету" по ходу выполнения процессов.

Адаптивный метод определения транзитивных зависимостей

В рассмотренном выше механизме логических часов зависимость между процессами устанавливается с помощью передачи одной единственной скалярной величины, что позволяет зафиксировать только прямую зависимость событий одного процесса от событий в других процессах. Поэтому для определения транзитивной причинно-следственной зависимости между событиями возникает необходимость для каждого процесса регистрировать каждое событие получения сообщения, т.е. обновлять и сохранять вектор прямых зависимостей в момент наступления такого события (или, по крайней мере, до наступления следующего события отправки сообщения в этом процессе). В противном случае транзитивную зависимость между событиями будет невозможно восстановить, т.к. цепочка прямых зависимостей окажется разорванной. Если в ходе распределенного вычисления процессы активно обмениваются сообщениями, требование регистрации каждого события получения сообщения может привести к необходимости сохранения большого объема данных и дополнительной нагрузке на процессы.

Данный метод позволяет адаптивно регистрировать только часть событий, т.е. сохранять информацию об их зависимостях, обеспечивая при этом возможность определять транзитивную причинно-следственную зависимость между зарегистрированными событиями. Другими словами, данный метод позволяет выделять непустое подмножество существенных (регистрируемых) событий $ER \subseteq E$ и фиксировать причинно-следственный порядок $\rightarrow R$, индуцированный исходным порядком $\rightarrow$ на множестве E. Для этого процессы по ходу своего выполнения должны отслеживать не только прямую зависимость, но и более длинные цепочки зависимостей между любыми двумя регистрируемыми событиями. Это приводит к необходимости пересылать в сообщениях больше информации, нежели несет в себе одна скалярная величина.

Поэтому адаптивный метод можно рассматривать как промежуточное решение между отслеживанием только прямой зависимости (когда с каждым сообщением передается всего один скаляр, но требуется регистрировать каждое

событие получения) и отслеживанием полной транзитивной зависимости в векторных часах (когда с каждым сообщением передается вся векторная отметка времени, полностью описывающая все зависимости передающего процесса).

Пример работы адаптивного метода приведен на рис. 3. Регистрируемые события отмечены жирным кружочком.

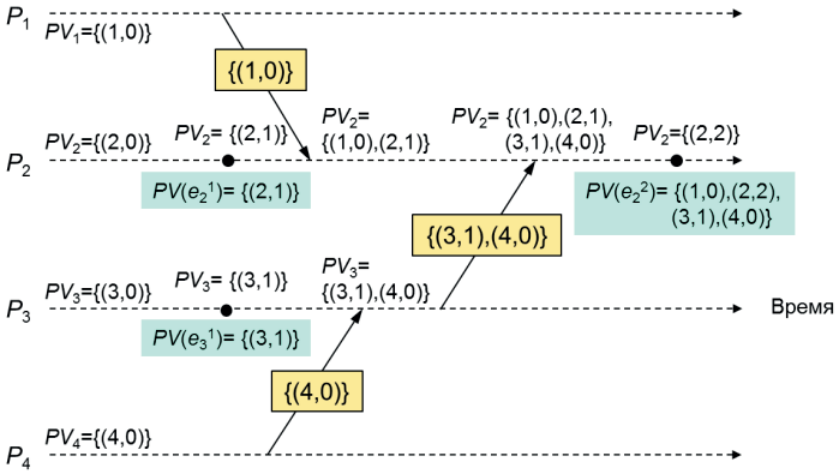

Рис. 3. Пример работы адаптивного метода.

Транзитивные зависимости заданного события ei могут быть восстановлены путем анализа отметок времени событий, с которыми ei связано отношением псевдо-прямой зависимости. Если такой анализ провести рекурсивно, то могут быть найдены все события, произошедшие раньше ei.

Основное преимущество метода заключается в том, что этот метод позволяет настраивать оптимальный размер отметки времени, передаваемой с каждым сообщением. А именно, количество пар, содержащихся в логических частично-векторных часах, может быть ограничено сверху путем регистрации дополнительных событий. Например, если требуется, чтобы размер логического времени не превышал k пар, достаточно, чтобы процесс зарегистрировал дополнительное событие, когда число накопленных им зависимостей достигнет k. Такое свойство адаптивной регистрации событий позволяет гибко контролировать накладные расходы на передачу сообщений при распределенном вычислении.

3.3. Задание на выполнение работы

Разработать программу, которая реализует векторные часы и моделирует их работу в распределенной системе.

Требования к программе:

- В программе должна быть реализована многопоточность для более реалистичного и эффективного моделирования распределенной системы.
- Потоки (узлы системы) должны обмениваться сообщениями.
- Каждый поток должен поддерживать свой вектор времени.
- Программа должна выводить логи событий (отправка, получение сообщений) и текущее состояние векторных часов для каждого потока.

3.4. Варианты заданий лабораторной работы

Индивидуальное задание на выполнение лабораторной работы, которое дополняет базовый вариант, студент получает от преподавателя.

Далее приводятся базовые варианты заданий на выполнение лабораторной работы.

Вариант 1. Реализовать классические векторные часы с использованием многопоточности.

- Каждый процесс должен выполняться в отдельном потоке, моделируя локальные события и отправку сообщений.
- Передачу сообщений производить, используя потокобезопасную очередь, обеспечивая безопасное взаимодействие между потоками.
- Выводить финальные векторные часы каждого процесса, что позволит проанализировать причинно-следственные зависимости.

Вариант 2. Реализовать оптимизированную передачу векторных часов, используя метод дифференциальной пересылки векторного времени.

- Каждый процесс должен выполняться в отдельном потоке, моделируя локальные события и отправку сообщений.
- Передачу сообщений производить, используя потокобезопасную очередь, обеспечивая безопасное взаимодействие между потоками.
- При отправке сообщения передавать только компоненты вектора, изменившиеся с момента последней отправки конкретному процессу.
- Реализовать логику обновления вектора получателя на основе полученных изменений.

Вариант 3. Реализовать механизм логических часов с фиксацией прямых зависимостей.

- Каждый процесс должен выполняться в отдельном потоке, моделируя локальные события и отправку сообщений.
- Передачу сообщений производить, используя потокобезопасную очередь, обеспечивая безопасное взаимодействие между потоками.
- Реализовать структуру данных для хранения вектора прямых зависимостей для каждого процесса.

- Написать функцию, которая может определить транзитивные зависимости для события, используя рекурсивный поиск по вектору прямых зависимостей.

Вариант 4. Реализовать систему частично-векторных часов для отслеживания зависимостей между событиями в многопоточном приложении.

- Каждый поток должен регистрировать события и отправлять сообщения другим потокам, сохраняя и передавая только необходимые зависимости.
- Передачу сообщений производить, используя потокобезопасную очередь, обеспечивая безопасное взаимодействие между потоками.
- В каждом потоке выводить информацию о текущем состоянии векторных часов после регистрации события и получения сообщения.

Вариант 5. Реализовать систему, которая отслеживает и восстанавливает транзитивные зависимости между событиями с использованием адаптивного метода.

- Каждое событие должно иметь помимо своего локального времени также список зависимых событий (т.е. транзитивных зависимостей).
- Передачу сообщений производить, используя потокобезопасную очередь, обеспечивая безопасное взаимодействие между потоками.
- Написать функцию, которая для заданного события восстановит все события, которые на него влияли (и для которых оно является зависимым).
- Для восстановления транзитивных зависимостей использовать рекурсивный анализ векторных часов.

3.5. Контрольные вопросы (темы)

1. Объясните, почему в распределенных системах невозможно использовать глобальные физические часы для упорядочивания событий.
2. Сформулируйте условия непротиворечивости логических часов. Как они связаны с причинно-следственным порядком?
3. Опишите алгоритм работы скалярных часов Лэмпорта. Какие правила применяются при отправке и получении сообщений?
4. Почему скалярные часы не являются строго непротиворечивыми? Приведите пример двух событий, которые параллельны, но имеют разные отметки времени.
5. Как с помощью скалярных часов реализуется линейное упорядочивание событий? Зачем нужны идентификаторы процессов в этом случае?
6. Дайте определение векторных часов. Как они позволяют определить параллелизм событий?
7. Почему размер вектора в векторных часах равен числу процессов в системе? Можно ли использовать вектор меньшей длины?
8. Объясните принцип дифференциальной пересылки векторного времени.

9. В чем недостаток часов, фиксирующих только прямую зависимость? Почему они не подходят для задач, требующих анализа транзитивных связей?
10. В каких сценариях дифференциальная пересылка векторного времени эффективнее базовой реализации?
11. Почему часы, фиксирующие прямую зависимость, неприменимы для задач отладки распределенных систем в реальном времени?

4. Лабораторная работа №2

«Алгоритмы на основе получения разрешений»

Продолжительность выполнения работы – 4 часа.

4.1. Цель работы

Целью лабораторной работы является приобретение навыков работы с основными алгоритмами координации распределенных процессов на основе получения разрешений на примере алгоритма Лэмпорта, алгоритма Рикарта-Агравалы и решения задачи «обедающих философов», а также проведение анализа работы данных алгоритмов в многопоточной среде.

4.2. Краткие теоретические сведения

Взаимное исключение в распределенных системах

Процессам распределенной системы часто приходиться координировать свои действия. Например, они могут соревноваться за возможность работы с разделяемыми ресурсами, требующими эксклюзивного доступа. Для организации такой работы, во-первых, нужно в коде каждого процесса выделять некоторую его часть, называемую критической секцией (англ. critical section), в которой есть обращения к таким ресурсам. Во-вторых, необходимо реализовать тот или иной механизм взаимного исключения (англ. mutual exclusion), выражающийся в удовлетворении следующего основного требования: во время выполнения критической секции одного процесса с обращениями к разделяемым ресурсам ни один другой процесс не должен выполняться в своей критической секции, работающей с любым из этих ресурсов.

Отсутствие общей памяти в распределенных системах не позволяет использовать разделяемые переменные (такие как семафоры) для решения рассматриваемой задачи: распределенные алгоритмы взаимного исключения должны опираться исключительно на обмен сообщениями между процессами. Разработка таких алгоритмов осложняется тем, что приходится иметь дело с произвольными задержками передачи сообщений и отсутствием полной информации о состоянии всей системы. Также не делается никаких

предположений об относительной скорости выполнения процессов и об их количестве.

Основные требования к алгоритмам взаимного исключения формулируются следующим образом.

Безопасность. В каждый момент времени в критической секции (КС) может выполняться не более одного процесса.

Живучесть. Каждый запрос на вход в КС рано или поздно должен быть удовлетворен.

Условие живучести алгоритма взаимного исключения говорит о том, что нельзя допускать ситуации взаимной блокировки, т.е. бесконечного ожидания прихода сообщения, которое никогда не придет, и ситуации голодания, т.е. бесконечного ожидания разрешения на вход в КС одним процессом, в то время как другие процессы неоднократно получают доступ к КС.

Отсутствие голодания также является условием справедливости. Другой контекст определения справедливости обслуживания запросов на вход в КС связан с установлением порядка вхождения процессов в КС и ограничением на число позволяемых входов-выходов в КС для одного процесса, пока другой процесс ожидает входа в КС. А именно, логично потребовать, что в случае, если один запрос на вход в КС произошел раньше другого, то и доступ к КС должен быть предоставлен в таком же порядке. Если механизм взаимного исключения удовлетворяет этому требованию, и все запросы на вход в КС связаны отношением причинно-следственного порядка, то ни один процесс не сможет войти в КС более одного раза, пока другой процесс ожидает получения разрешения на вход в КС.

Все многообразие алгоритмов взаимного исключения в распределенных системах на самом деле базируется на реализации одного из двух основных принципов.

1. Алгоритмы на основе получения разрешений (англ. permission-based algorithms).

В этом случае для входа в КС процессу требуется собрать "достаточное" число разрешений от других процессов распределенной системы. Свойство безопасности будет выполнено, если такое число разрешений сможет получить лишь один процесс из всех процессов, желающих войти в КС. Свойство живучести обычно обеспечивается за счет упорядочивания запросов по их отметкам логического времени или с помощью ациклического графа предшествования, в котором вершины соответствуют процессам распределенной системы, а направленные ребра описывают приоритет процессов друг над другом для определения порядка предоставления доступа к КС.

2. Алгоритмы на основе передачи маркера (англ. token-based algorithms).

Для таких алгоритмов право войти в КС материализуется в виде уникального объекта – маркера, который в каждый момент времени может содержаться только у одного процесса или же находиться в канале в состоянии пересылки от одного процесса к другому. Свойство безопасности в этом случае будет гарантировано ввиду уникальности маркера. Поэтому основные усилия

при разработке алгоритмов данного класса направлены на обеспечение свойства живучести, а именно, на управление перемещением маркера таким образом, чтобы он рано или поздно оказался в каждом процессе, желающем войти в КС. Существуют два подхода к решению этой задачи: непрерывное перемещение маркера среди всех процессов распределенной системы (лат. perpetuum mobile) или перемещение маркера лишь в ответ на получение запроса от заинтересованного в нем процесса (англ. token asking methods). Отметим, что при формировании запросов на владение маркером необходимо создать такие условия их распространения, при которых каждый запрос рано или поздно достигнет процесса, в котором находится маркер, вне зависимости от перемещений самого маркера.

Централизованные и распределенные алгоритмы.

Централизованный алгоритм предполагает, что основная часть работы выполняется одним процессом или небольшой группой процессов (в сравнении с их общим числом), в то время как остальные процессы играют незначительную роль в достижении общего результата. Обычно эта роль сводится к получению информации от главного процесса или предоставлению ему нужных данных. Поэтому централизованный алгоритм всегда является асимметричным: различные процессы выполняют логически разные функции, хотя, возможно, в чем-то и совпадающие. Наиболее подходящей для централизованных алгоритмов архитектурой вычислительных систем является архитектура клиент-сервер, в которой сервер владеет и распоряжается информационными ресурсами, а клиент имеет возможность воспользоваться ими.

В распределенных алгоритмах каждый процесс играет одинаковую роль в достижении общего результата и в разделении общей нагрузки. Поэтому распределенные алгоритмы являются симметричными: все процессы исполняют один и тот же код и выполняют одни и те же логические функции. На самом деле абсолютно симметричные алгоритмы представляются практически идеальной конструкцией и для реализации в реальных системах более предпочтительными являются частично распределенные алгоритмы, обладающие естественной асимметрией в функциях своих узлов.

Алгоритмы на основе получения разрешений. Алгоритм Лэмпорта.

Алгоритм, предложенный Л. Лэмпортом, является одним из самых ранних распределенных алгоритмов взаимного исключения и призван проиллюстрировать применение скалярных часов для линейного упорядочивания операций, производимых процессами в распределенной системе, таких, например, как вход в критическую секцию и выход из КС.

В его основе лежат предположения о том, что все каналы связи обладают свойством FIFO, и сеть является полносвязной. Данный алгоритм обобщает централизованный алгоритм, в котором запросы на вход в КС помещаются в очередь и обслуживаются из нее по порядку, в том смысле, что позволяет рассматривать такую очередь в виде отдельного объекта, разделяемого между

всеми процессами, а не находящегося под управлением одного единственного процесса (координатора).

Важным преимуществом алгоритма Лэмпорта является также то, что запросы на вход в КС обслуживаются не в произвольном порядке, как в централизованном алгоритме, а в порядке их возникновения в системе, т.е. в порядке, определяемом их отметками логического времени. Поэтому, если один запрос на вход в КС произошел раньше другого, то и доступ к КС по первому запросу будет предоставлен раньше, чем по второму. Здесь под отметками времени запросов на вход в КС мы будем подразумевать упорядоченные пары (Li , i), где Li – скалярное время процесса Pi на момент формирования запроса, i – идентификатор процесса Pi.

Алгоритм Лэмпорта опирается на принцип: чтобы реализовать представление общей совместно используемой очереди, каждый процесс работает с ее локальной "копией" и для определения единого для всех процессов порядка обслуживания запросов использует их отметки логического времени, упорядоченные линейно. Чтобы каждый запрос оказался в каждой из таких локальных "копий", всякий раз, когда процесс собирается войти в КС, он помещает свой запрос вместе с отметкой времени в свою локальную очередь и рассылает сообщение с этим же запросом всем остальным процессам.

При получении запроса остальные процессы помещают его уже в свои локальные очереди. По порядку обслуживания запросов процесс получит право войти в КС, когда значение отметки времени его запроса окажется наименьшим среди всех других запросов. Остается лишь ответить на вопрос: как процессу убедиться в том, что его запрос имеет наименьшую отметку времени?

Действительно, из-за задержек передачи сообщений, возможна ситуация, когда в локальных очередях двух различных процессов в течение некоторого временного интервала наименьшей отметкой времени будут обладать разные запросы, что может привести к нарушению требования взаимного исключения.

Прежде чем процесс примет решение о входе в КС исходя из состояния своей собственной очереди, он должен получить от всех других процессов сообщения, гарантирующие, что в каналах не осталось ни одного сообщения с запросом, имеющим время меньшее, чем его собственный запрос. Благодаря свойству FIFO каналов связи и правилам работы логических часов такими сообщениями могут быть ответные сообщения, высылаемые при получении сообщения с запросом, или же любые другие сообщения, направляемые процессу, запрашивающему вход в КС, с отметкой времени большей, чем отметка времени его запроса.

Обратим внимание, что представленный алгоритм является распределенным: все процессы следуют одним и тем же правилам, и каждый процесс принимает решение о входе в КС только на основе своей локальной информации. Также отметим, что, когда получение всех сформированных запросов на доступ к КС подтверждено всеми процессами, все локальные очереди будут пребывать в одинаковом состоянии, что как раз и позволяет рассматривать их как "копии" некоторой отдельной очереди, разделяемой между

процессами. Поэтому можно сказать, что решение на вход в КС принимается на основе локальной информации, которая, однако, глобально согласована.

Алгоритм Лэмпорта обладает свойствами безопасности и живучести.

Чтобы оценить эффективность работы алгоритма Лэмпорта, отметим, что для обеспечения взаимного исключения необходимо переслать 3(N – 1) сообщений: для входа в КС требуется (N – 1) сообщений REQUEST и (N – 1) ответных сообщений REPLY, для выхода из КС требуется (N – 1) сообщений RELEASE.

Алгоритм Рикарта-Агравалы

Алгоритм Рикарта-Агравалы (Ricart–Agrawala Algorithm) является усовершенствованной версией алгоритма Лэмпорта и относится к классу алгоритмов на основе получения разрешений. Он используется для обеспечения взаимоисключения в распределенных системах без использования централизованного узла или токена. Алгоритм предполагает полностью распределенный подход, где процессы координируют доступ к критической секции (КС) посредством обмена сообщениями.

В отличие от алгоритма Лэмпорта, в котором требуется три фазы обмена сообщениями, алгоритм Рикарта-Агравалы требует только 2(N-1) сообщений вместо 3(N−1), что повышает его эффективность.

Описание среды выполнения

Рассматриваемая система состоит из N процессов P1,P2,...,PN, каждый из которых является независимым вычислительным узлом в распределенной системе. Процессы могут быть расположены на разных машинах и взаимодействуют между собой только путем передачи сообщений по сети. Эти процессы не имеют общего глобального времени и не обладают общей памятью, что делает проблему синхронизации особенно сложной.

Каждый процесс выполняет некоторую локальную работу и в определенные моменты времени запрашивает доступ к критической секции. Критическая секция — это участок кода, к которому в один момент времени должен иметь доступ только один процесс, чтобы избежать состояния гонки (race condition) и обеспечить корректность данных. Примерами критической секции могут быть:

- Запись в общий файл или базу данных.
- Обновление глобальной переменной, используемой несколькими процессами.
- Выполнение транзакции в распределенной системе.

Для корректного функционирования системы необходимо соблюдение строгого порядка выполнения критической секции, что требует реализации механизма взаимоисключения.

Алгоритм Рикарта-Агравалы удовлетворяет следующим ключевым требованиям:

1. Свойство взаимного исключения (Mutual Exclusion)

- В любой момент времени не более одного процесса может находиться в критической секции.
- Если один процесс уже выполняет КС, все остальные должны ожидать.

2. Свобода от взаимоблокировок (No Deadlocks)
   - Система не должна попадать в состояние, когда процессы ждут друг друга бесконечно, но никто не получает доступ к ресурсу.
3. Свобода от голодания (No Starvation)
   - Если процесс запрашивает доступ к критической секции, он гарантированно получит его через конечное число шагов.
   - Исключается ситуация, когда один процесс постоянно получает доступ, а другие — нет.
4. Согласованный порядок доступа (Fairness & Ordering)
   - Запросы должны обрабатываться в хронологическом порядке согласно логическим часам Лэмпорта.
   - Если процесс Pi отправил запрос раньше процесса Pj, то Pi должен получить доступ к КС первым.

Формальная модель работы процессов

Пусть каждый процесс Pi управляет локальными логическими часами Ti, которые увеличиваются при каждом важном событии (например, отправке или получении сообщения). Для обеспечения согласованного порядка процессов мы используем частично упорядоченные метки времени из часов Лэмпорта:

$$Ti=Ti+1$$

Когда процесс отправляет запрос на вход в критическую секцию, он приписывает этому запросу текущую метку времени Ti. Все процессы должны согласованно обрабатывать эти метки времени, чтобы решить, кто должен получить доступ к КС.

Каждый процесс ведет свою очередь запросов, содержащую поступившие запросы от других узлов. Очередь поддерживается в упорядоченном виде по следующим правилам:

- Сначала обрабатываются запросы с меньшими метками времени.
- Если два запроса имеют одинаковую метку времени, приоритет получает процесс с меньшим идентификатором.

Таким образом, при наличии двух запросов:

REQUEST(T1,P1)

REQUEST(T2,P2),

если $T1<T2$, то процесс P1 получает доступ первым. Если $T1=T2$, то процесс с меньшим идентификатором получает доступ.

Глобальные допущения и ограничения

Предполагается, что все сообщения, переданные между процессами, достигают адресата и не теряются в сети. Если сообщение теряется, могут возникнуть взаимоблокировки, поэтому необходимо использовать механизмы подтверждения доставки.

Алгоритм не предполагает, что процессы могут аварийно завершаться во время выполнения критической секции. В случае сбоя процесс может "повиснуть", что приведет к зависанию всей системы.

Нет приоритетов у разных процессов: все запросы обрабатываются в порядке поступления.

Описание алгоритма

Алгоритм использует отметки времени из часов Лэмпорта и обмен сообщениями между процессами.

Каждый процесс взаимодействует с остальными через следующие сообщения:

- REQUEST(Timestamp, Process_ID) – запрос на вход в КС.
- REPLY(Process_ID) – подтверждение возможности входа.

Каждый процесс поддерживает:

- Логический временной счетчик T, обновляемый согласно часам Лэмпорта.
- Очередь запросов, упорядоченную по временным меткам.

Алгоритм включает три фазы:

Фаза 1: Запрос на вход в критическую секцию

Если процесс Pi хочет войти в КС:

- Увеличивает свои логические часы:

$$Ti=Ti+1$$

- Отправляет REQUEST(Ti, Pi) всем остальным процессам.
- Добавляет свой запрос в локальную очередь.

Фаза 2: Обработка запросов

При получении REQUEST(Tj, Pj) от процесса Pj, процесс Pi выполняет:

- Увеличивает свои часы:

$$Ti=\max(Ti,Tj)+1$$

- Если он сам не хочет войти в КС или его запрос имеет более высокий временной штамп (Ti>Tj), то он немедленно отправляет REPLY.
- В противном случае он задерживает ответ и отправит его только после выхода из КС.

Фаза 3: Вход в критическую секцию

Процесс Pi может войти в КС, если:

- Он получил (N-1) REPLY сообщений от всех остальных процессов.

– Его запрос находится в начале его локальной очереди.

После выполнения критической секции:

– Удаляет свой запрос из очереди.
– Отправляет отложенные REPLY тем процессам, которым он не ответил.

Достоинства алгоритма:

– Полностью распределенный подход, не требующий центрального сервера.
– Эффективнее алгоритма Лэмпорта (меньше сообщений).

Недостатки алгоритма:

– Все узлы должны быть в сети и отвечать, иначе возможны задержки.
– Возможны задержки в обработке при высоком количестве запросов.

Задача «обедающих философов»

Задача «обедающих философов» (Dining Philosophers Problem) была предложена Эдсгером Дейкстрой в 1965 году и является классической проблемой синхронизации в параллельных вычислениях. Она иллюстрирует конкуренцию процессов за ограниченные ресурсы, демонстрируя потенциальные проблемы взаимоблокировки (deadlock), голодания (starvation) и справедливого распределения ресурсов.

Формулировка задачи

– В задаче участвуют N философов, сидящих за круглым столом.
– Между каждой парой философов лежит по одной вилке (всего N вилок).
– Философ может либо размышлять, либо есть.
– Для еды философ должен взять две вилки одновременно – слева и справа от себя.
– После еды он кладет вилки обратно на стол и снова начинает размышлять.

Цель алгоритма — обеспечить согласованное поведение философов, чтобы избежать:

1. Взаимоблокировки.

Взаимоблокировка возникает, если все философы одновременно возьмут по одной вилке (например, сначала возьмут левую), а затем попытаются взять вторую, которая уже занята соседом.

Решение:

– Использовать асимметричный алгоритм: одни философы берут сначала левую вилку, а другие — правую. Это нарушает симметрию и снижает вероятность взаимоблокировки.
– Ввести централизованный диспетчер, который выдает разрешение на еду, обеспечивая, чтобы одновременно ели не более N−1 философов.

2. Голодания.

Голодание может возникнуть, если философы с разной частотой получают доступ к вилкам, и некоторые из них остаются без пищи на неопределенно долгий срок.

Решение:

- Использовать алгоритм честного распределения, например, очередь или справедливый планировщик.
- Ограничить максимальное время размышления и приоритетное выделение ресурсов философам, которые давно не ели.

3. Состояния гонки (неконтролируемого доступа к ресурсам).

Если два философа одновременно пытаются взять одну и ту же вилку, может возникнуть некорректный доступ к ресурсу.

Решение:

- Использовать мьютексы (mutex) для блокировки вилки.
- Ввести атомарные операции при захвате вилки.

Варианты решения задачи «обедающих философов»

1. Централизованный диспетчер (семафор)

- Один процесс управляет доступом к вилкам.
- Философы получают разрешение только от диспетчера.
- Минус: узкое место — если диспетчер выходит из строя, система зависает.

2. Использование мьютексов

- Каждая вилка — это мьютекс, который философ должен захватить перед использованием.
- Проблема взаимоблокировки решается нарушением симметрии.

3. Решение Дейкстры (ресурсная иерархия)

- Все ресурсы (вилки) нумеруются.
- Философ всегда берет сначала вилку с меньшим номером, а затем с большим.
- Это исключает циклы ожидания, предотвращая взаимоблокировку.

Пример решения задачи «обедающих философов» с использованием мьютексов (mutexes), устраняя взаимоблокировки и обеспечивая корректное поведение системы на языке Python:

```
import threading  # Импортируем модуль для работы с потоками
import time       # Импортируем модуль для работы со временем
import random     # Импортируем модуль для генерации случайных чисел

# Класс, представляющий вилку
class Fork:
    def __init__(self):
        self.lock = threading.Lock()  # Создаем мьютекс для блокировки вилки

    def pick_up(self):
        self.lock.acquire()  # Захватываем мьютекс (вилка занята)

    def put_down(self):
        self.lock.release()  # Освобождаем мьютекс (вилка доступна)

# Класс, представляющий философа
class Philosopher(threading.Thread):
    def __init__(self, index, left_fork, right_fork):
        threading.Thread.__init__(self)  # Инициализация потока
        self.index = index               # Уникальный номер философа
        self.left_fork = left_fork       # Вилка слева от философа
        self.right_fork = right_fork     # Вилка справа от философа

    def run(self):
        while True:  # Бесконечный цикл размышлений и еды
            self.think()  # Философ размышляет
            self.eat()    # Философ ест

    def think(self):
        print(f"Философ {self.index} размышляет...") # Выводим статус философа
        time.sleep(random.uniform(1, 3))  # Ждем случайное время (1-3 секунды)

    def eat(self):
        # Решение проблемы взаимоблокировки: нарушаем симметрию
        # Философ с четным индексом берет сначала левую вилку, потом правую
        # Философ с нечетным индексом берет сначала правую вилку, потом левую
        if self.index % 2 == 0:
            self.left_fork.pick_up()  # Берем левую вилку
            self.right_fork.pick_up()  # Берем правую вилку
        else:
            self.right_fork.pick_up()  # Берем правую вилку
            self.left_fork.pick_up()  # Берем левую вилку

        print(f"Философ {self.index} ест...")  # Выводим статус философа
        time.sleep(random.uniform(1, 2))  # Философ ест случайное время

        # После еды философ кладет вилки обратно на стол
        self.left_fork.put_down()
```

```
        self.right_fork.put_down()

# Основная программа
num_philosophers = 5  # Количество философов
forks = [Fork() for _ in range(num_philosophers)]  # Создаем вилки

# Создаем философов, каждый из которых получает две вилки (слева и справа),
# обеспечиваем циклическое соединение вилок по кругу
philosophers = [Philosopher(i, forks[i], forks[(i+1) % num_philosophers])
                for i in range(num_philosophers)]

# Запускаем потоки (философы начинают размышлять и есть)
for p in philosophers:
    p.start()
```

4.3. Задание на выполнение работы

Разработать программу, в которой реализован один из основных алгоритмов координации распределенных процессов на основе получения разрешений.

Требования к программе:

- В программе должна быть реализована многопоточность для более реалистичного и эффективного моделирования распределенной системы.
- Потоки (узлы системы) должны обмениваться сообщениями.
- Программа должна выводить логи событий (отправка, получение сообщений) и текущее состояние для каждого потока.

4.4. Варианты заданий лабораторной работы

Индивидуальное задание на выполнение лабораторной работы, которое дополняет базовый вариант, студент получает от преподавателя.

Далее приводятся базовые варианты заданий на выполнение лабораторной работы.

Вариант 1. Реализация алгоритма Лэмпорта для взаимного исключения.

- Каждый процесс должен выполняться в отдельном потоке, моделируя локальные события и отправку сообщений.
- Каждый процесс должен уметь:
  - Отправлять запрос на вход в критическую секцию.
  - Получать запросы от других процессов и отправлять подтверждения.
  - Входить в критическую секцию только после получения подтверждений от всех других процессов.
  - Обновлять логические часы при обмене сообщениями.
- Реализовать многопоточное взаимодействие через очередь.

Вариант 2. Реализация алгоритма Рикарта-Агравалы для взаимного исключения.

- Каждый процесс должен выполняться в отдельном потоке, моделируя локальные события и отправку сообщений.
- Каждый процесс должен уметь:
  - Отправлять запрос на вход в критическую секцию всем остальным процессам.
  - Получать запросы от других процессов и сохранять их в очередь приоритетов, используя логические часы для упорядочивания.
  - Отправлять разрешение (REPLY) другим процессам, если их запрос имеет более высокий приоритет.
  - Входить в критическую секцию, когда получены все разрешения от остальных процессов.
  - Покидать критическую секцию и уведомлять другие процессы, позволяя им продолжить выполнение.
- Реализовать многопоточное взаимодействие через очередь.

Вариант 3. Решение задачи «обедающих философов».

- Программа должна быть реализована с использованием многопоточности, где каждый философ является отдельным потоком или задачей.
- Каждый философ должен поочередно выполнять три действия:
  - Думать (время ожидания, симуляция процесса без ресурсов).
  - Брать вилки (если оба ресурса доступны).
  - Есть (время обработки пищи, симуляция использования ресурсов).
- Реализовать различные стратегии решения задачи, в соответствии с заданием, полученным от преподавателя, анализируя их эффективность и устойчивость к взаимоблокировкам (deadlocks) и состояниям голодания (starvation).
- Программа должна быть устойчивой к состоянию голодания (когда один философ никогда не может взять вилки).
- Каждый философ должен выводить в консоль или файл лог сообщений, описывающих его действия (например, «Философ 1 берет вилки», «Философ 2 ест» и т.д.). Логи должны содержать временные метки для отслеживания порядка выполнения действий.

4.5. Контрольные вопросы (темы)

1. В чем суть алгоритмов на основе получения разрешений?
2. Как работает алгоритм Лэмпорта для взаимного исключения?
3. Как алгоритм Лэмпорта гарантирует порядок выполнения критической секции?
4. Чем отличается алгоритм Рикарта-Агравалы от алгоритма Лэмпорта?
5. Какие условия должен выполнить процесс в алгоритме Рикарта-Агравалы, чтобы войти в критическую секцию?

6. Почему алгоритм Рикарта-Агравалы требует меньше сообщений, чем алгоритм Лэмпорта?
7. В чем заключается проблема взаимной блокировки в задаче обедающих философов?
8. Какие методы используются для предотвращения голодания процессов в задаче обедающих философов?
9. Какие структуры данных используются для хранения запросов в алгоритме Лэмпорта?
10. Какие основные свойства должны соблюдать алгоритмы на основе получения разрешений (например, справедливость, отсутствие взаимоблокировки)?
11. В каких реальных системах могут применяться алгоритмы на основе получения разрешений?

5. Список рекомендуемой литературы

1. Бабичев С. Л., Коньков К. А. Распределенные системы: учебное пособие для вузов / С. Л. Бабичев, К. А. Коньков. — Москва: Издательство Юрайт, 2019. — 507 с.
2. Гниденко И.Г., Морозов С.К., Федоров Д.Ю. Многопроцессорные системы и параллельное программирование. СПб.: Санкт-Петербургский государственный экономический университет, 2019. — 68 с.
3. Романов А.А. (сост.) Распределенные вычисления и приложения. Учебное пособие. — Ульяновск: УлГТУ, 2018. — 151 с. (Понятия распределенных вычислений и распределенной системы.)
4. Косяков М. С. Введение в распределенные вычисления. СПб.: НИУ ИТМО, 2014. — 155 с.
5. Дейтел Х. М., Дейтел П. Дж., Сантри С. И. Технологии программирования на Java 2. Распределенные приложения; Бином-Пресс - Москва, 2011. - 464 с.
6. Радченко Г.И. Распределенные вычислительные системы – Челябинск: Фотохудожник, 2012. – 184 с.

6. Образец титульного листа для оформления отчетов

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

ОТЧЁТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ № __

по дисциплине «Модели и технологии распределенных вычислений»

Вариант № __

Выполнил:

Студент группы ИС

Иванов И.И.

Проверил:

К.т.н., доцент каф. ВМКСС

Гаранин С.А.

Москва - 2025